Hardware Based Approach To Confine Malicious Processes From Side Channel Attack

By

Zirak Allaf

Supervisor

Dr. Mo Adda

This thesis is submitted in partial fulfilment of
the requirements for the award of the degree of
Doctor of Philosophy of the University of Portsmouth.

January, 2018

I would like to dedicate this thesis to my loving family …

The soul of my Father and Father-in-law, *whose memories persist in being forever*

my Mother, *whom I owe my life*

my Mother-in-law, *who always encourages me*

my loving Wife "Sivar", *who was being my guardian during my PhD course*, and

children "San & Lara, *who are my eternal gratitude*

List of publications

Published articles

Allaf, Z., Adda, M., and Gegov, A. TrapMP: Malicious Process Detection By Utilising Program Phase Detection. In 2019 International Conference on Cyber Security and Protection of Digital Services(Cyber Security). IEEE.

Allaf, Z., Adda, M., and Gegov, A. (2018). Confmvm: A hardware-assisted model to confine malicious vms. InUKSim2018: UKSim-AMSS 20th International Conference on Modelling Simulation. IEEE.

Allaf, Z., Adda, M., and Gegov, A. (2017). A comparison study on flush+reload and prime+probe attacks on aes using machine learning approaches. In UK Workshop on Computational Intelligence, pages 203–213. Springer.

Allaf, Zirak. "Review of data leakage attack techniques in cloud systems." Data Security in Cloud Computing. IET, 2017. IET Digital Library. http://digital-library.theiet.org/. September 2017.

Declaration

Whilst registered as a candidate for the above degree, I have not been registered for any other research award. The result and conclusions embodied in this thesis are the work of the named candidate and have not been submitted for any other academic award.

April 2019

Acknowledgements

Firstly, I would like to take this opportunity to give thanks to all those who have made a contribution to the accomplishment of my challenging PhD journey.
I would like to sincerely thank my supportive supervisor, Dr. Mo Adda, for his great help, continuous encouragement and guidance during my study. My special thanks also go for my second supervisor Dr. Alexander Gegove for his great support and advice.
I would like to express my sincere gratitude to my beloved father *Mohammad Allaf* who was an ideal example of fatherhood and a great example of a loyal person.
I am indebted to my good friends Dr. Twana Haji, Dr. McCalpin John, Dr. Rasber Rashid and Dr. Karwan Qader for the excellent suggestions they had given me during my study.
Finally, my special thanks go out to my family: my mother and mother-in-law, my beloved wife (Sivar), son and daughter (San, Lara), my sisters Rupak and Dilpaq, my brothers, and nephews and nieces, for their continuous love and support throughout my studies. This challenge would not have been accomplished without them.

Abstract

Cryptography can be considered as a set of algorithms which primarily relies on mathematical theories with computational supports to be practised in computer systems. Therefore, Cryptography is employed as the main component to security solutions mainly in Internet and could computing. Despite this, hardware and firmware implementations have failed to securely manage program executions in computational environment. This limitation has made it possible for hackers to carry out side channel attacks on computer systems and steal sensitive cryptographic components, such as the secret keys, which are used in securing communication channels. Such issues are alarming, and crucial, and therefore obligate the detection and identification of attackers of the systems.

In this thesis, side channel attacks, exploiting the weakness in hardware and firmware implementations, are addressed along with existing counter-measures. The current side-channel attack techniques show that attackers can exploit the micro-architecture vulnerabilities to achieve their goals. The recent Meltdown attack for instance misuses program execution attributes such as "out-of-order execution", through a Flush and Reload mechanism, to break the logical isolation between the memories of two independent processes in the kernel space.

Furthermore, in this work, a real-time detection and identification framework has been developed against side-channel attacks. The concept behind this is to take a course of program phase analysis to extract Malicious Loop (ML) phases at the processor core level. Unlike previous works, the proposed detection system within the framework does not rely on synchronisation between the attackers and the victim. Instead, it banks on the Hardware Performance Counters (HPC) utilisation, which is a hardware feature built-in to the modern computational environments. The framework offers high accuracy and efficient detection of Flush+Reload activities before the attacker completes the malicious task. Moreover, the detection can be achieved with minimum time required to detect the attack(s) in both native and cloud systems at the same cost. Additionally, the framework benefits from very low overhead performance approximately less than 1

Table of contents

List of figures

List of tables

Nomenclature

Abbreviations

AES Advanced Encryption Standard

APIC Advanced Programmable Interrupt Controller

ASLR Address Space Layout Randomisation

AUC Area Under Curve

CART Classification and Regression Trees

CAT Cache Allocation Technology

CL Control line

CV Cross-Validation

DoS Denial of Service

FR Flag Registers

GPR General Purpose Registers

HPC Hardware Performance Counters

IaaS Infrastructure as a Service

IOAPIC Inpu/Output Advanced Programmable Interrupt Controller

IR Instruction Register

KASLR Kernel Address Space Layout Randomisation

LAPIC Local Advanced Programmable Interrupt Controller

LLC Last Level Cache

MAR Memory Address Register

MBR Memory Buffer Register

ML Malicious Loop

ML Malicious Loop

MSR Model Specific Registers

MWA Moving Window Aggregation

NN Neural Network

OS Operation System

PMC Model Specific Registers

PMU Performance Monitor Unit

PMU Performance Monitoring Unit

PTE Page Table Entry

ROC Receiver Operating Characteristic

RSA Rivest–Shamir–Adleman

RTS Real-Time System

SGX Software Guard Extensions

SPEC Standard Performance Evaluation Corporation

SSE Sum of Squared Error

SVM Support Vector Machine

TR Temporary Register

TSC Time Stamp Counter

VT Virtualization Technology

Symbols

MW	Molecular weight [kg/kmol]

Subscripts

1	Inlet

Chapter 1

Introduction

Cloud computing is a client server configuration that uses services which are available on the Internet to enable its users to access technologies with no need for the users to understand either the technologies or the services themselves that allow their delivery. The range offered by cloud computing models means that the cloud provides a computational environment of great richness through which the users can access the applications that they require at any time and wherever they may be. The hardware resources can be scaled up as required and the cloud offers unparalleled flexibility, allowing for a quick response to the user's requirements with no intervention by the users' IT managers. Set against this is the drawback that the ease with which the cloud can be accessed increases the possibility of threats, both to resources that are being shared with others and to the computing environment itself. The essential point of the cloud is that it makes hardware and software services available to users in a way that allows the applications to be continuously available to meet the end-user needs. However, the integrity and protection of data becomes a key issue, particularly when, as it is usually the case, the data is manipulated and the hardware is outsourced by a third party, the cloud provider, in a location which is not disclosed to the end user. In brief, the data manipulation and hardware security are outside of the control of the data owners and the data, which for most companies and organisations are critical asset, - are vulnerable to data leakage attacks.

This chapter will set out the importance of this research and will provide the aims and objectives of the research in addition to the research questions.

1.1 Motivation

The internet in general, and cloud computing in particular, is increasingly in use on a variety of devices, both desktop and mobile. These devices outsource the end users' privacy to the

cloud. As a result of this, the data is exposed to a number of threats, and maintaining the data and keeping it safe from attackers is a challenging task. The most important role in maintaining data security in all of the states that it passes through (during its transmission, its processing and its storage) falls to cryptography. The vulnerability of the data attracts hackers who seek, through side channel attacks, to obtain access to the data by stealing important components of cryptography, one of which is secret keys. Previous research has put forward a number of sophisticated techniques through which side channel attacks can be carried out. Success makes the protection of the user data sufficiently unreliable such that the end users either avoid using cloud services or restrict their use of cloud services. Recent research has demonstrated that considerable use has been made of the opening for side channel attacks and that these openings have allowed attackers to retrieve the entirety of a security key in less than a minute on native systems and less than three minutes in cloud systems (Irazoqui et al., 2014) and through various system settings (Irazoqui et al., 2015). This matters because there have been some well publicised hacks in which the hackers have stolen large amounts of data. Such attacks offer the ability to read the arbitrary memory in which sensitive information might be stored. In addition, recent research showed that side channel attacks affect any end-users who use devices including PC, laptop, servers, tablets and mobiles, which support Intel, AMD and embedded processors. Furthermore, the attacks do not care about any particular system or platform such as Windows, XOS and any UNIX-based OSes, as cloud systems including Docker containers Xen and OpenVP are affected by the attacks. Meltdown attacks are a case study which is where an attack can break down the logical isolation mechanisms between the different applications which are supposed to be securely managed by the OS. Some of the existing detection and protection software available such as anti-viruses fail to detect side channel attacks because the attack does not leave any traces through traditional log files. This is due to the fact that the side channel attack code runs in user land, does not require any privileges and does not require system calls during the program execution. However, researchers in previous studies have showed there to be various detection (Briongos et al., 2017) and prevention (Kim et al., 2012) techniques, and patches (Simakov et al., 2018), in both native and cloud systems. Each response has limitations, such as the detection applying only in its native OS (Alam et al., 2017; Payer, 2016), monitoring all VM instances, monitoring all processes in the systems (Payer, 2016), monitoring suspicious malicious VMs and monitoring sensitive programs such as cryptography-based applications (Zhang et al., 2016a) or the detection rely on the synchronisation between the attacker and victims (Kulah et al., 2018). It is therefore critical to develop an efficient and reliable framework which is able to detect and identify side

channel attacks in native and cloud systems at the same cost, each of which requires a very low number of samples to function accurately. This results in being able to maintain the performance of the system while the proposed detection system is running.

1.2 Research Questions

This section provides the hypothesis of this study and sets out the research questions.
Hypothesis: *A complex computational environment can be modelled and analysed automatically in relation to security as defined by the user.*

The following research questions will be posed in order to validate the hypothesis.

1. How feasible is it to create a new knowledge-based framework which is capable of mitigating side channel attacks launched against cryptography algorithms?
2. Can supervised Machine Learning be used to build a classification model which is capable of detecting side channel attacks? If yes, is it possible to achieve optimum accuracy when detecting such attacks?
3. Is it possible to use the automated observation of processor cores to isolate the activities which are identifying processes with malicious intent?
4. What countermeasures can be used effectively to mitigate data insecurity and the risks that it poses to the user's sensitive data on both native and cloud systems?
5. What impact on the occurrence of monitored events can be attributed to the CPU workloads running independently in the computing environment?

1.3 Research Aim and Objectives

The aim of this research is to create a new knowledge-based framework which is able to leverage hardware support in order to analyse the process activities with the aim of detecting malicious processes which may be running in the user space. The framework should be able to mitigate and eliminate security threats against cache memories, particularly when the cryptography algorithms are running. Machine Learning techniques will be used as part of this process. Adding tree-based classifiers takes the research into the search for techniques to mitigate complex security problems. Following objectives had to be fulfilled to achieve this aim:

1. Propose the utilisation of Machine Learning methods as Neural Network, Decision Trees, Random forest and k-NN in order to provide a comparison of their efficiency and accuracy under different workloads. This work is represented in Chapter 3.

2. Through an examination of CPU component usage, examine user programs' execution attributes to extract program phase of malicious programs.

3. Propose a new mechanism for process identification in order to detect the attackers when malicious processes are present in the system to avoid performance overhead. This work is represented in Chapter 4.

4. Propose minimum hardware-assisted PMU custom settings to observe the process activities in the CPU, which will also help to determine other security vulnerabilities. This work is represented in Chapter 4.

5. Suggest that different attacks may have behaviours which are reliant on different events. This work is represented in Chapter 4.

6. Describe the micro-architecture of modern CPU in the way that it permits comprehension of the attack as well as the Performance Monitor Units (PMU) which are capable of supporting the detection models of side channel attacks. This work is represented in Chapter 3 and 4.

7. Use Hardware Performance Counters (HPC) to provide an overview of the program's execution in the user space. This work is represented in Chapter 3 and 4.

8. Examine the program phase analysis which emerges detection and identification of malicious processes. This work is represented in Chapter 4.

9. Analyse the machine learning techniques introduced in side channel attack detection by different studies. This work is represented in Chapter 2 and 4.

10. Design and implement a model which incorporates the analysis of the detection techniques to make accurate and reliable side channel attack detection. This work is represented in Chapter 4.

11. Evaluate the accuracy of the detection system by testing it with SPEC CPU 2006 benchmark suit. This work is represented in Chapter 4.

12. Deploy the detection system on the host OS and evaluate the energy efficiency and performance of the host OS along with the detection system operations. This work is represented in Chapter 4.

1.4 Chapter Outlines

1. **Chapter Two** presents a discussion of the existing literature on side channel attacks from their first use against cryptographic algorithms until now. Vulnerable points in hardware and software that have been exploited by attackers to steal sensitive information will be described, as well as a number of existing countermeasures for mitigation and prevention of side channel attacks. Points for and against existing countermeasures will be discussed. Finally, research gaps and limitations in the literature will be itemised.

2. **Chapter Three** sets out the background detail required to understand typical CPU architecture in a manner descriptive of the intercommunication between cache memories and the CPU in a way that it reinforces the description of side channel attack techniques. The state of data and its importance will be described, as will be the core technologies making the cloud vulnerable to side channel attacks. Instrumentation used in both attack and defence will be described. This chapter also demonstrates the ability of machine learning methods to detect malicious loops that can be used for side channel attacks (Flush+Reload and Prime+Probe), which rely on synchronisation between the attacker and victim.

3. **Chapter Four** sets out the challenges and motivations and the key intuition on which this study's approach is based. The framework design will also be explained. This chapter also explores a range of machine learning techniques and addressed the limitations of single decision tree algorithms in the context of our work. Finally, This chapter details the illustration of the identification phase, including a brief background of interrupt handlers and the way they work with HPCs.

4. **Chapter Five** summarises the research, draws conclusions and suggests fruitful avenues for future studies.

Chapter 2

Background and Related Work

A number of studies have shown it is possible to carry out side channel attacks on CPU components and sensitive applications through the use of user credentials and compromising technologies. This chapter, firstly, provides a necessary background on the current technologies which manage and transfer digital information from front to back end services and how they prone to side channel attack. Secondly, summarises the literature from the previous studies into the use of techniques and mechanisms both to conduct side channel attacks and to counteract them. Also explored is the attitude of the hacker and their intention in the use of a number of approaches for exploitation of up software and hardware, with a focus on components that attackers have targeted and compromised. At the end of the chapter, we review the gaps in research into presently available countermeasures and discuss those countermeasures' limitations, then address them and link them with the sections of the thesis.

2.1 Background

This section charts the vulnerabilities to leakage in CPU architecture, technologies, and data and describes current side channel attacking techniques and the countermeasures available to defeat them.

Weaknesses in hardware and software can be exploited by using leakage of information in a channel-based attack to generate communication between two processes that share physical resources but should not communicate with each other; the environment may be virtualised or non-virtualised.

The lion's share of such attacks has been shown in recent studies to target cryptosystems through the exploitation of poor software standards and by improper use of physical resources in a shared environment. What causes attacks on cryptosystems is that they are necessary

components in data protection in a client/server and cloud environment (as mentioned in 2.2.3) as data passes through untrusted communication channels and/or is processed in untrusted computing environments. The heavy reliance of such environments on co-residency and shared features raises the likelihood that attackers will find themselves next to a target's neighbour under virtual isolation, increasing vulnerability to computational analysis.

Where side channel attacks take place on native systems where cloud computing is becoming mainstream, and the attacks are low level, it becomes clear that side channel attacks are easy to mount in the Cloud.

2.1.1 Data State and Vulnerabilities

Clearly, data is one of any organisation's most important assets. To understand the dangers inherent in data leakage, a definition is needed of exactly what is meant by "data" in computing environments. According to Shabtai et al. (2012) data passes through three stages in the processing cycle: Data-At-Rest (DAR), Data-In-Use (DIU) and Data-In-Motion (DIM). These three stages embrace everything that happens to data from the moment of its creation through its processing (which may involve a number of computing resources) to the point at which it is transmitted through the cloud. The hardware resources in which data may be found include files, the content of memory, and packets transmitted over a network, and the representation of the data in each of these resources is different. As Table 1 shows, there have been data leakage attacks in recent years in every one of the possible data states (Chen et al., 2010; Ristenpart et al., 2009; Stolfo et al., 2012; Yarom and Falkner, 2014), The subsections following contain explanations of data states and the threats applicable at each of these states. The main focus will be on data leakage attacks while the data is in the in-use stage (DIU).

2.1.1.1 Data-At-Rest

In the DAR state, data is in storage. A computer's hard disk constitutes a permanent data storage, and DAR is the stage at which the data is in such a device. One concern that arises in connection with cloud computing is transparency of the data's location as a result of the way that the complexity of the infrastructure making up the cloud is hidden from cloud consumers – a cloud consumer is not aware of where their data is stored. Some organisations considering migrating to the cloud hesitate as a result of this lack of transparency, which causes them concern about who may be dealing with data that is sensitive to them. There are also concerns on the part of consumers about the action of others who may be co-located on the same storage device with malicious intent. (Ristenpart et al., 2009) indicated the

possibility of a covert channel attack, hard disk-based, against arbitrarily selected VMs on the Amazon cloud facility, EC2. They supported this suggestion by successfully identifying co-residents in cloud storage and demonstrating the transmission of data between VMs when accessing shared storage devices.

Concerns also arise over the way that, to counteract the limits on space available in main memory, the operating system will deal with large files by loading part of the file (with contiguous pages) into main memory while the rest of the file stays on the system's storage device where it has the same physical addresses as those from which the file originated. Bernstein (2005) showed how this feature could be exploited in the construction of a covert channel attack that would use time variation to identify those parts of the file already loaded and those still on the disk. This works because it takes less time to access recently accessed parts of the file than to access those that have not been touched recently.

2.1.1.2 Data-in-motion

DIM (Data-in-motion) describes the data as it is transferred over a network from source to destination. In this condition, data is at risk from eavesdropping attacks which can inspect packets. Cryptographic technology such as Secure Shell (SSH) has been developed to protect the data at this point in its cycle, but the data is not fully protected against information leakage. Song et al. (2001) demonstrated an example of a leakage attack on packages being carried on the network – specifically in that case, looking for the password. That study developed a Hidden Markov Model (HMM) and an algorithm to predict the sequence of key presses by the target, the keys in question being the password.

Software as a Service (SaaS), is using cloud systems to enable the delivery of applications through the web to end users. The data while in the DIM stage will be client/server requests and responses, and needs protection from eavesdropping attacks. The protection will be in the form of encryption, but that is also not a complete protection against hostile attacks, as attackers will still be able to analyse the packet size and timings in order to extract sensitive information (Chen et al., 2010).

2.1.1.3 Data-in-Use (DIU)

The Data-in-Use (DIU) state occurs when data has been loaded into a computing resource. For the most part, the resource in question will be a component of a CPU – perhaps a register, perhaps a cache, perhaps main memory. When data is loaded into a CPU register from a storage device, it will normally have to pass through a number of hierarchical

buffers which will include main memory and CPU caches (L1, L2 and L3) before it reaches the registers. Once there, it is prepared for read/write operations (among other possible operations). Depending on the application in question, the data alignment and organisation can vary according to data type as it is loaded into main memory. Possibilities include lists, linked lists, arrays, class, and structure. Physical resources in multitasking environments are shared between processes that occur in different layers. Examples would include the OS layer for page sharing and the application layer for shared libraries. The data will be used by those resources frequently, and so the operating system alternates the use of those resources by processes so that a memory region is shared between processes – an AES buffer lookup table would be an example (Bernstein, 2005). Research recently has shown that data in multitasking systems is vulnerable in the DIU stage in both the OS layer (e.g. page sharing) (Gruss et al., 2015a; Suzaki et al., 2011), the achieved memory deduplication attack, and the application layer (e.g. shared library) Irazoqui et al. (2014). The main focus in this study is on the DIU state.

2.2 Core Technology Vulnerabilities in Cloud Systems

Cloud computing is a combination of existing technologies including the web, cryptography and virtualisation to provide companies and organisations and enterprises with optimum solutions. Technologies may be wide-ranging in the domains they serve, with the health sector, education, government, social networks and e-Commerce all well represented. The fact remains that combining technologies in cloud-based data handling introduces vulnerabilities capable of being exploited by attackers seeking to steal sensitive data for malicious purposes. A list will now be provided of common cloud technologies that have been compromised in data leakage attacks.

2.2.1 Web Technology

In SaaS (Software-as-a-Service), applications are delivered online to web clients. Web technology is a client/server configuration that enables communication over networks. Most desktop applications have today been converted to web applications thanks to their ease and low installation cost, user-friendly interface, and no update required from the client. The platform is independent and continually being technologically enhanced. All the end user needs is a web browser to provide communicate with the server. The drawback is that this communication takes place through networks, which carry the danger that they may

reveal sensitive information such as data, application states and state-transits. Encryption is needed to protect this information from network sniffers, but this does not prevent the theft of information. Chen et al. (2010) showed the continuing viability of sensitive information leakage.

2.2.2 Virtualization Technology

Virtualisation Technology (VT) begins with actual physical hardware resources (CPU, RAM, IO and network) and from them creates a number of virtual machines (VMs), each of which appears to have (is visualised as having) a physical presence with its own hardware and operating system. Each VM is ascribed appears logical isolation and appears to be an independent part of the system. Each VM runs on top of an additional layer called the hypervisor, of which the chief responsibilities are to monitor and manage shared resources between VMs through a sandboxing technique in such a way as to maintain security. VT amplifies discrete hardware resources to serve many VMs. The primary feature is a memory sharing memory technique, which has been widely developed by VT-based software designers for greater memory efficiency; an example is Kernel Same Page (KSM) in KVM. This does, though, significantly impact the security of the system, particularly in regard to data leakage attacks. Opportunities to build hidden communication channels make it possible for nefarious VM operators to exploit hardware vulnerabilities, leading to information leakage in which data exchange is performed through unauthorised and illicit processes. Security threats are endemic to the VM system, which is especially susceptible to data leakage attacks.

2.2.3 Cryptography

Cryptography is in effect a set of algorithms relying primarily on mathematical theories computationally supported for use in computer systems. Cryptography as the chief component in a number of security solutions is designed to be compromised, It is used widely in a number of domains as a data protection solution against third parties planning to steal credential information with malicious intent. Application may be multi-use by the operating system layer or application layers, and it is suitable for a variety of purposes including email services, banking, health records, and encrypted stored data.

As an OS-level solution, it may protect data by storing it in encrypted form on a physical storage device. During installation of a modern operating system, an optional step is to request encryption of user files before they are stored on an internal storage device. This facility gives cloud consumers reliability because need not be concerned about transparency of storage

location. It is also usable at the application layer on top of the OS layer. Especially for web applications, software designers embed cryptographic algorithms into pages to encrypt data exchanged between client/server in response to a request in order to guard against eavesdropping (Chen et al., 2010). This can have value in cloud-based Platform-as-a-Service (PaaS) applications.

Although widely used in computer and cloud systems, cryptography remains vulnerable to information leakage attacks. This study is focused on side channel attacks in native and cloud systems, mounted to extract sensitive information like secret keys by exploiting vulnerabilities in software implementation and/or hardware architecture, rather than through a brute force attack based on guesswork or an attack on the algorithms' underlying implementation.

2.3 Side-channel Attacks

Current trends especially in cloud systems, are towards shared systems, with the result that securing execution time and shared resources have become major issues.

Recent research has shown that side channel attacks are not restricted to cryptography. In fact, they are used to target data in such environments as: Database (Kellaris et al., 2016) smart card (Messerges et al., 1999, 2002) BTree search algorithm (Dachman-Soled et al., 2017) satellite (Santhanam et al., 2017), CAPTCHA (Hernandez-Castro and Ribagorda, 2010), printer (Backes et al., 2010), Web Applications (Chen et al., 2010), keystroke (Cai and Chen, 2011; Lipp et al., 2017) and etc.

Side channel attacks have been mounted against cryptosystems to extract secret keys using information sources instead of through Brute Force attacks or attacks against algorithms' mathematical implementation, in which the attacker relies on being able to deduce the target's secret keys to obtain information, identifying those keys as non-functional properties of program output. Methods include observation of such things as execution time, memory usage in shared systems, and the timing and scale of encryption cycles). Popular algorithms researched in the last ten years include RSA (Yarom and Falkner, 2014)(Aciiçmez, 2007), AES (Osvik et al., 2006) , DES (Acıiçmez et al., 2010). Attackers monitor shared resources to collect the fine detail information that will help identification of the usage of targeted data by victims. The attacker does not need to be privileged in order to obtain yield the information, as resource sharing will suffice to enable the attack.

Attackers utilise C/C++ due to the direct access to the hardware resources such as memories in the system. Moreover, they are the language most operating systems and kernel components are written in. Further, they are able to run assembly language directly with

native language's statements. Besides, they are the programming language most suitable for areas where data leakage attack models exist. In addition, they are able to virtualise memories into array form, especially CPU cache memories, giving greater capabilities to programmers while dealing directly with such on-board resources.

Side channel attacks were first introduced by (Kocher, 1996). Over the past 15 years, there has been extensive research into side channel attacks extracting cryptographic keys through CPU caches (L1, L2 and L3 or LLC) which have been practised successfully on both native and cloud systems. The approach has been to analyse the access time variations by tracing use of the CPU cache

The first practical attack against a cryptographic algorithm DES was proposed by Tsunoo et al. (2003) in 2003, taking some 10^{23} samples and retrieving ca 90% of the secret key.

Researchers first looked at the AES algorithm in 2005. It was practised first by Bernstein (2005); with the attacker remotely analysing the algorithm's overall execution time. Bernstein also published a full implementation of the attack, which was subsequently extended and refined by Neve et al. (2006). They addressed limitations of Bernstein's attack and retrieved all key bits though taking fewer samples. In a 2013 enhancement of Bernstein's side channel attack, Aly and ElGayyar (2013) reproduced the attack on modern CPUs with the latest version of AES.

As cloud computing has become more popular, so has cryptography, with the focus of researchers being to solve weaknesses of logical isolation between cloud entities existing on the same physical machine. Ristenpart et al. (2009) addressed Amazon EC2 cloud systems' internal hardware vulnerabilities with proposed attacks at high level and low resolution. In 2012, (Zhang et al., 2012) greatly improved resolution through the use of L1 cache, while in 2014, Yarom and Falkner (2014) achieved a side channel attack of very high resolution through the use of L3 combined with unrelated processes, each of which was on a different core. LLC has since become the hardware most heavily targeted by attackers seeking to extract secret keys (Gruss et al., 2015b)(Irazoqui et al., 2015).

A number of side channel attack techniques exist; the four most often used by researchers, especially in cloud computing, are:

Time+Evict (Osvik et al., 2006)(Tromer et al., 2010):This technique assumes that a shared library is linked at the same time to programs run by attacker and target; an example would be a lookup table in AES. Each party can access the lookup table. The attacker monitors cache line(s) synchronised with the array[1] with the aim of first finding the average time taken by a single encryption, after which the attacker triggers encryption and evicts

[1]The lookup table is represented as an array when it is loaded in to main memory

those cache lines that have already touched the array. The attacker then triggers a series of encryption, measuring each one. Any encryption call that takes longer than average is an indication that the target recently accessed the evicted cache line(s).

Prime+Probe (Zhang et al., 2012)(Maurice et al., 2015b): This technique involves monitoring by an attacker of the target process by filling the CPU cache with the attacker's data. The purpose of this is to identify attacker's cache lines evicted by the target's data. To achieve this, the attacker uses a malicious loop which, in each iteration, sleeps for a specific time to wake the attacker process and measure access time variations to its cache lines. A longer access time shows the cache line to have been evicted by the target so that it must be reloaded from a higher memory level. This technique works on all memory levels.

Flush+Reload (Gullasch et al., 2011)(Yarom and Falkner, 2014): This is the inverse of the Prime+Probe technique. Both attacker and target need to be able to access the same data concurrently (the shared library feature mentioned in 1.5.2.6). The attacker process flushes the cache to ensure removal of all cache lines. After flushing, the attacker waits until data is accessed by the target. The flush means that data accesses must be from higher level memory, and the target's data fills some cache lines. The attacker then tries to access data from the same source as the target; a shorter access time means that the target has already accessed the cache line in question

Flush+Reload has become an powerful technique in achieving side channel attacks against various machine learning algorithms, exploiting OS, hardware and application vulnerabilities, and various platforms x86 and mobile devices Lipp et al. (2016)

Flush+Flush (Gruss et al., 2015b)(Gruss et al., 2015c): Two flushes comprise this technique, which differs from Flush+Reload in that it does not have a reload step. The cache is thus free of misses. The attacker is relying on time variations in a series of flushes rather than monitoring cache line accesses

2.4 Side Channel Attacks in Two Decades

There has been widespread use of several attack techniques in leaking sensitive data in cloud systems. They are similar to techniques already described, except in the use of different system settings and CPU architecture.

As already mentioned, one of the things covert and side channel attacks offer attackers is the ability to interfere with underlying hardware activities and, for example, measure changes in the time it takes to access higher level memory. This is the mechanism at the centre of such attacks, allowing an attacker to assume the target's credentials through resource sharing.

What follows is a generic model for Cloud system attacks, along with the history of the past 15 years of attacks against a variety of shared resources.

Cloud components' vulnerabilities to data leakage attacks has already been shown. Achieving data leakage depends on how attackers view the secret keys or other targets they are out to steal. The importance of cryptographic algorithms in real systems as a way of encrypting data has already been mentioned, and has encouraged deployment of sophisticated attack techniques to obtain secret keys. This attack model's scope will be the targeting of secret keys. Two steps are required for a data leakage attack to succeed: placement a malicious VM, and the attack itself in cloud systems.

When the attacker has identified the data type, the next step is to locate the attack processes on the same physical machine as the target. Placing a new VM instance in a lab is cheaper than in such real systems such as EC2 (Ristenpart et al., 2009)(Xu et al., 2011) because, in real systems, cloud providers try to hide cloud infrastructure's complexity as well as the complexity of data storage to prevent the cloud being mapped. The attacker therefore needs further action to happen before the target can be located Ristenpart et al. (2009) overcame this problem by establishing a covert channel attack in Amazon EC2, a real cloud system, by using network probing techniques to reveal EC2 mapping in which internal and external network address spaces correspond to an entity creation. This helped the attacker in the early stages to find the internal map of EC2 in order to locate the attack process in the same EC2 zone as the target. In this way, the possibility is created that the attacker and the target may be on the same physical server in the same zone.

Next, the attacker uses such attack techniques prime+prob and Flush+Reload (2.4). The attacker monitors the target's activities on shared hardware. There has been considerable recent research into these attacks across a range of on-board resources including CPU caches (L1 (Zhang et al., 2012), L2 (Wang et al., 2012) and L3 or LLC (Yarom and Falkner, 2014)) and memory pages (Zhang et al., 2011). Table 2.1 shows typical attacks mapped against physical resources.

The theft of secret keys from cryptographic algorithms relies on the nature of the algorithms. AES algorithm, for example Bernstein (2005)(Briongos et al., 2016) encrypts plain text by means of a lookup table which holds values to be used during the encryption rounds. As the most critical of the algorithm's components, this table has been targeted by attackers. AES attackers look for cache contentions and cache line accesses to determine recent cache line usage, generating candidate elements in the target's lookup table by utilising auto timing registers.

In the RSA algorithm (Yarom and Falkner, 2014), however, dependence on the S-Box

is replaced by dependence on the mathematical operations square and multiply. Attackers therefore seek to trace execution of the target's program rather than memory accesses.

To investigate attacks against L1 cache Zhang et al. (2012) built constructed a side channel attack using L1 instruction cache to extract AES secret keys from a target VM co-resident with the attacker's VM, with both VMs running libgcrypt shared library. They addressed hardware and software noise sources against the attacking VM and could reduce the noise during the observation stage by combining SVM with the Hidden Markov Model (HMM) to deduce key bits.

To investigate attacks against L2 cache, Ristenpart et al. (2009) introduced a cross-VM covert channel attack against CPU L2 cache, targeting large files to transfer messages between two VMs with the object of identifying co-resident VMs on shared storage devices. Their interested was in hard disk contention patterns revealed by recording variations between VMs in access time to certain portions of the files. Xu et al. (2011) improved resolution of this attack through higher bandwidth and lower error rate.

Attackers then improved the leakage attack model through the use of L3 and other resources, recovering entire keys in reduced times. Previous studies had shown core-sharing between attack and target VMs to be a requirement of the attacks. Previous to 2014, L1 and L2 cache levels were most targeted and it was necessary to pair attack and target processes on the same core. Zhang et al. (2012) attack model used IPI interrupts to force core migration of the target process, allocating it to the same core as the attacker process. Hoevery, Yarom and Falkner (2014) introduced a new form of side channel attack by using L3 cache and exploiting the inclusiveness feature. Their proposal was for a Flush+Reload technique to extract from the GnuPG RSA implementation the private key's components, amounting to some 97.7% of the key. In this case, attack and target processes were on different cores but in shared page settings. The authors successfully constructed an LLC-based channel between two unrelated processes in a virtualised environment.

Deduplication is a key function in virtualisation, enabling the host to reclaim large amounts of memory where contents are identical. Previous studies showed this feature to have been exploited in cloud systems. The reason for Microsoft Azuru disabling hyperthreading in the cloud system was the side channel attack proposed by Marshall et al. (2010). Suzaki et al. (2011) proposed matching as a technique to identify applications (sshd and apache2) running on Linux OS and Firefox and IE6 on Windows XP) and to discover a targeted file downloaded by a browser on the target VM. Bosman et al. (2016) suggested a side channel attack against Windows Edge Browser based on a Java script to retrieve an HTTP hashed password. This vulnerability encouraged CPU makers to disable the feature, but this has

not prevented the leaking of information against hardware resources by attackers. Irazoqui et al. (2015) tried to overcome the leakage of information but, even when deduplication is disabled, it remains possible. Pages of huge size were used, and the attacker was able to obtain complete physical addresses from virtual addresses. In cache addressing, the same physical addresses are used for cache addressing.

In shared page settings, an LLC based channel was successfully created between two unrelated processes in a virtual environment. Deduplication is a key function in virtualisation, enabling the host to reclaim large amounts of memory where content is identical.

Table 2.1 Side and Covert Channel Attack Classifications

Data Leakage Attack Techniques in Cloud Systems							
Stat	Type	Tech	App	Res	Type	System	Publications
DAR	CC	PP	–	L1	File	IaaS	(Ristenpart et al., 2009)
	CC	PP	–	L1	File	IaaS	(Xu et al., 2011)
	SC	–	DES	L1	Key	Native	(Tsunoo et al., 2003)
	CC	FR	–	L1,2	–	mobile	(Lipp et al., 2016)
	CC	–	–	–	–	native	(Kocher et al., 2011)
	CC	–	–	–	–	native	(Messerges et al., 2002)
	CC	–	–	–	–	native	(Picek et al., 2017)
	CC	–	–	–	–	native	(Maurice et al., 2015a)
	CC	–	–	–	–	native	(Aciicmez and Seifert, 2007)
	CC	–	–	–	–	native	(Brickell et al., 2006)
	CC	ET	–	LLC	–	native	(Brasser et al., 2017)
	CC	ET	–	LLC	–	native	(Yarom et al., 2015)
	CC	ET	–	LLC	–	native	(Yarom et al., 2015)
	CC	–	–	DRAM	–	native, cloud	(van der Veen et al., 2016)
	CC	FR	–	DRAM	–	native, cloud	(Pessl et al., 2016)
	SC	FR	KASLR	–	resolve address	native, cloud	(Gruss et al., 2016a)
	–	–	–	PTE	–	native	(Gruss et al., 2016b)

Continued on next page

Table 2.1 – continued from previous page

Data Leakage Attack Techniques in Cloud Systems							
Stat	Type	Tech	App	Res	Type	System	Publications
	–	–	–	PTE	–	native	(Seaborn and Dullien, 2015)
	CC	PP	RSA	L1, L2	key	native	(Gruss et al., 2017b)
	CC	PP	RSA	L1, L2	key	native	(Percival, 2005)
	SC	PP	AES	L1	Key	native	(Bernstein, 2005)
	CC	SMT/FU	program	L2	file	native	(Wang and Lee, 2006)
	SC	SMT-cache	AES	L2	key	native	(Wang and Lee, 2006)
	SC	–	RSA	L1I	Key	native	(Aciiçmez, 2007)
	SC	ET, PP	AES	Page	–	native	(Osvik et al., 2006), (Tromer et al., 2010)
	SC	PP	–	L1I	–	native	(Acıiçmez et al., 2010)
	CC	PP	browser	Pages	file	PaaS	(Chen et al., 2010)
	SC	ET, PP	AES	L3	key	native	(Gullasch et al., 2011)
	CC	PP	–	L2	file	cloud	(Xu et al., 2011)
	CC	PP	browsers	pages	file	IaaS	(Suzaki et al., 2011)
	CC	PP	browsers	pages	file	IaaS	(Suzaki et al., 2011)
	SC	PP	AES	L2	key	native	(Gullasch et al., 2011)
	SC	FR	AES	L3	key	native	(Irazoqui et al., 2014)
	CC	PP	browsers	pages	file	PaaS	(Zhang et al., 2014)
	CC	FF	RSA	L3	Key	IaaS	(Yarom and Falkner, 2014)
	SC	FR	AES	L3	Key	IaaS	(Irazoqui et al., 2015)
	CC	FF	browser	L3	file	IaaS	(Gruss et al., 2015b,c)
	SC	FR	Kernel Module	KASLR	–	native, IaaS	(Hund et al., 2013)
	SC	PP	KASLR	L1	–	native, IaaS	(Evtyushkin et al., 2016)
	SC	FR	AES	L3	key	native	(Gülmezoğlu et al., 2015)

Continued on next page

Table 2.1 – continued from previous page

Data Leakage Attack Techniques in Cloud Systems							
Stat	Type	Tech	App	Res	Type	System	Publications
	SC	PP	AES	L2	key	IaaS	(Younis et al., 2015)
	SC	FR	AES	LLC	key	native	(Gruss et al., 2015d)
	SC	FR	kernel	LLC	Memory	native, cloud	(Hund et al., 2013)
	SC	PP, FR	AES	LLC	key	native	(Kayaalp et al., 2016)
	SC	PP, FR	AES	LLC	key	native, IaaS	(Gulmezoglu et al., 2016)
	SC	–	browser	–	JPEG	IaaS-SGX	(Xu et al., 2015)
	SC	–	browser	–	JPEG	native	(Jana and Shmatikov, 2012)
	SC	RAM	browser	RAM	JPEG	native	(Gruss et al., 2016b)
	SC	RAM	browser	–	JPEG	native	(van der Veen et al., 2016)
	SC	PP	Btree	LLC	–	native	(Dachman-Soled et al., 2017)
	SC	FR, PP	Btree		keyst-rokes	embed-ded	(Lipp et al., 2017)
	SC	FR, PP	RSA	cache-bank	key	native	(Yarom et al., 2017)
	SC	PP	RSA	page-fault	key	native	(Brasser et al., 2017)
	SC	PP	AES	L1	key	IaaS	(Moghimi et al., 2017)
	SC	FR	RSA	—	key	—	(Bernstein et al., 2017)
	SC	FR	RSA	BLISS, BLISS-b	key	—	(Pessl et al., 2017)
	SC	FR	RSA	BLISS	key	—	(Bruinderink et al., 2016)
	SC	FR	ECDH	–	key	—	(Genkin et al., 2017)
	CC	Timing	database	–	Dataset	IaaS	(Stolfo et al., 2012)
DIT							Continued on next page

Table 2.1 – continued from previous page

Data Leakage Attack Techniques in Cloud Systems							
Stat	Type	Tech	App	Res	Type	System	Publications
	CC	–	database	–	Dataset	IaaS	(Stolfo et al., 2012)
	CC	RSA	database	–	Dataset	IaaS	(Acıiçmez et al., 2007)
	SC	–	browser	–	http	native	(Qian et al., 2012)

2.4.1 Targeted Data Types

The following subsections will describe commonly targeted data types that can be compromised by side channel attacks. Table 2.1 shows how side and covert attacks are classified by the way they target data types against cryptographic applications and web browsers in native and cloud systems.

2.4.1.1 Cryptographic Keys

A key is a set of letters or symbols forming a string that, when cryptographic algorithms are used to combine it with plain text, produce cipher text; the process also works in reverse. This is the core source of cryptographic encryption and decryption. Cryptographic keys may be 64-bit, 128-bit, 256-bit, or some other size. A short key might be the right choice against a Brute Force attack, because modern CPUs can have a large number of cores and it may be possible to find the right key by checking the maximum number of possible keys in a short time while, even with the power of the modern CPU, a long key takes a long time. Example: it would take 150 trillion years to crack a 128-bit AES key (Penchalaiah and Seshadri, 2010)).

Keys come in different types, and to use them it is necessary to know how the algorithm works. An AES algorithm encrypts and decrypts by use of a secret key, while the RSA algorithm uses a private and a public key, one to encrypt and one to decrypt. An encryption algorithm starting the generation of cyphertext by a combination of a key with plain text meets a number of mathematical operations in sequence (the operations being, for example, division, multiplication and mode). The algorithms most commonly used for data protection in real systems are AES and RAS – but they are also the most attacked algorithms. Attackers suborn the algorithms' components and characteristics to get the results they want. AES, for example, generates encrypted data through the use of T-Table, while RSA uses mathematical operations. While the algorithms are executing cryptographic operations, an organised utilisation of the CPU components. Thus, the observation of the memory transactions of the processes which belong to the algorithms can be made easily.

2.4.1.2 Files

Features like shared storage and service synchronisation which are everywhere in the Cloud lead Cloud users to get in the habit of storing documents remotely on storage over which they have no control; examples are Google Drive, Dropbox, and SkyDrive, which share similar data methods (Chu et al., 2013). Shared characteristics of this sort can be seen in online applications deliverable as PaaS. Recent research has looked at the weaknesses in implementing the web applications that can lead to a data leakage attack. Percival (2005); Ristenpart et al. (2009); Xiao et al. (2012); Xu et al. (2011) all targeted user shared storage files by building side/covert channels through shared resources.

2.4.2 Source of Leakage

There are a number of characteristics offered by cloud technologies to offer highquality services, such as resource pooling, multi-tenancy and rapid elasticity (Mell and Grance, 2011). However, they are vulnerable to information leakage attacks, particularly resource sharing features, which are fundamental for cloud computing to gain sufficient performance to lease sufficient number of cloud tenants. Consequently, studies have shown that this feature is prevalently exploited by attackers for their malicious intentions.

Stealing sensitive data is possible at all data states (DAR, DIP and DIT) (see Table 2.1) as they all have special consideration and techniques to achieve the attacks. Data in multitasking/users' systems can be processed by different resources, such as CPU caches, memory, storage and network media. Each media has its own characteristics on which the attackers rely during experiments. So, it is crucial to focus on the common features that have already been exploited by previous studies

2.4.2.1 CPU Architecture

Past studies showed that CPU is the most targeted resources by attackers. It is the main physical resource in computational models due to interconnection with on-board resources such as main memory and IO devices through buses. Modern CPUs have multiple cores to offer more efficient performance and facilitate to accommodate a sufficient number of programs concurrently. Each core represents a logical isolated processor inside the CPU die. The CPU has different cache levels.

In computer systems, the size and speed of memories are ordered from small and high to large and slow (e.g. registers, L1, L2, L3 and main memory). The main responsibility of CPU caches is to buffer data requested from the main memory, due to the trade-off

performance between fast-to-slow and small-to-large memories in order to supply the CPU when it is operating on the requested data. Each core can have its own L1 and L2 caches, or only L1 privately, while L2 is shared between two or more cores, depending on the CPU architecture design, and cores from L3 and above are shared for all CPU cores. Ge et al. (2016) demonstrated modern microprocessors' architecture and their compromising to information leakage attacks in fine-grain details.

CPU cache is the main source of information that attackers relay on to perform side and covert channel attacks. By reviewing such attacks since last 15 years, it can be noticed that the attackers utilised one of the cache levels as primary communication channels between two processes (attack and victim) that provides the state of the victim process unintentionally. Earlier attacks targeted L1 and L2 cache as communication channels between attack and victim processes, which reside on the same core. However, core migration makes noise to the measurement when operating system alternates assigning processes to a core (Zhang et al., 2012). Researchers then continued to explore faster and higher bitrate attacks against L1 in cross-core settings. Until 2014, unified cache L3 has gained most of the attention due to higher resolution with less time required to recover sensitive information (Yarom and Falkner, 2014).

2.4.2.2 Main Memory

Rowhammer is much used as an attack on main memory, DRAM, and depends on a memory fault during program execution allowing an unauthorised program launched by the attacker to change bits of DRAM cells. The OS uses these bits in the management of memory locations. Repeated accesses of DRAM cells allow the attacker to make changes in adjacent memory locations (Gruss et al., 2017b). by means of which the attacker is able to deduce the target's memory location. A number of approaches have been used to launch side channel RowHammer attacks. Gruss et al. (2017b) used the Flush+Reload technique through exploitation of prefetch address translation (Gruss et al., 2016a) to map virtual to physical addresses, enabling attackers to map virtual addresses of attacker and victim to the same physical page. Pessl et al. (2016) proposed a high-speed covert channel up to 2Mb/s using a side channel attack across the CPU, without memory being shared. The authors bridge between two processors by way of main memory.

2.4.2.3 Timing

Most published side and covert channel attacks have relied on timing for their success. Attackers are interested in hardware activities at the most basic level including the number of cycles it takes to access a single line in L1 cache. Attackers use their own program and data to measure the differences in time taken to access memory locations, so that they can synchronise target data stored in the same memory in shared hardware. Modern processors such as Intel offer hardware support, a register to capture time of all operations with high accuracy. Intel also offers a Read Time Stamp Counter (RDTSC) instruction to read the counter register's value[2]. Attackers measure memory accesses using the RDTSC instruction through a number of techniques including Prime+Probe (Maurice et al., 2015b)(Percival, 2005), Flush+Reload (Yarom and Falkner, 2014) and FLush+Flush (Gruss et al., 2015c).

2.4.2.4 CPU Power Consumption

Attackers monitor the power consumption of the CPU over time in order to work out mathematical operations, on the basis that operations like multiplication and division use more CPU components than add and subtraction. This information is helpful to an attacker seeking to extract secret keys from cryptographic algorithms. This works because in a computer running cryptographic algorithms the CPU executes a series of divisions and multiplications increasing the CPU's power consumption. The attacker is then able to analyse power consumption variations to deduce the secret key bits (Banerjee et al., 2015; Wu et al., 2010). This approach will not be covered in the present study.

2.4.2.5 Page Sharing

A heavily used technique is memory page sharing, which finds widespread use in OSs and hypervisors like KVM Kernel Same Page (KSP) (8) and ESX Transparent Page Sharing (TPS) (Arcangeli et al., 2009) The OS or hypervisor scans memory pages looking for those in which the contents are identical in each time period, so that only one copy is kept and the rest are removed (Pan et al., 2011; Waldspurger, 2002). In virtualisation, a hypervisor running the same OS in multiple VMs can reclaim sufficient memory. VMWARE VMWARE (2009) statistically showed that, if ten VMs are running, the memory saving can exceed 40%.

Take, for example, a ten page shared by two VMs (VM1 and VM2). If VM1 modifies two pages, the OS executes a copy-on-write creating two private copies of the pages which

[2]RDTSC is an assembly instruction, it can be used in C and C++ in-line assembly. For more detail on how to use RDTSC in modern Intel CPUs (Paoloni, 2010)

are then referred to VM1. VM2 can no longer access VM1's private copies. This means that each VM has two private and eight shared pages. Variations in writing time between shared and private pages are observable by VM1. Writing on a shared page takes longer than on a private page, with the result, as shown by previous studies, that covert channel attackers can exploit this feature in computer (Wang and Lee, 2006) and cloud systems (Bosman et al., 2016; Suzaki et al., 2011; Xiao et al., 2012).

2.4.2.6 Shared Library

Shared libraries are code compiled to be linkable in run time by other programs. Loading a shared library into memory causes multiple linked programs to share the same memory locations. Memory is thereby saved because, instead of each program needing its own copy, only one copy is required. Shared libraries also have the advantage of easy maintenance, since a modified library, once loaded, is instantly available to all linked programs. There is, though, an exploitable vulnerability since a shared library shares characteristic between processes with a negative impact on data protection. Shared library vulnerability has been illustrated at its most visible in the use of the OpenSSL implementation of AES (Bernstein, 2005; Zhang et al., 2012). This provides a dynamic shared library `libcrypto` for linking with multiple programs in UNIX-based OSs. AES uses a lookup and S-Box table, an array of values for use during encryption rounds. If this table is used by a target during encryption, an attacker can see which elements of the lookup table the target has recently looked up.

2.4.2.7 Kernel Address Space Layout Randomisation

Kernel Address Space Layout Randomisation (KASLR) KASLR is a layer used by host OSs to prevent an unprivileged side channel attacker from being able to work out memory accesses on sensitive data, but Gruss et al. (2016a) exploited the prefetch instruction mechanism to map virtual memory addresses to physical addresses, giving the attacker information about hierarchical pages.

2.4.3 Types of Channel attacks

Side channel and covert channel attacks have been implemented recently as two common types of information leakage attacks. They look for hierarchical memory accesses by analysing variations in time taken by the accesses. These two types of attacks produce similar results, though covert channel attacks are broader than side channel attacks, and deployable in a number of layers including network (Shah and Blaze, 2009) (Hovhannisyan et al., 2015),

OS (Hund et al., 2013), I/O (Shah et al., 2006)(Ristenpart et al., 2009) and application (Gruss et al., 2015a). Side channel attacks specifically target cryptosystems in order to extract secret data through a number of routes including CPU components and OS vulnerabilities. The following sections will describe covert channel attacks but mostly side channel attacks.

2.4.3.1 Covert-Channel Attacks

The idea of a side channel attack began with the work of Lampson (1969). in late 1969 who pointed out that communication channels between two processes could be created for the exchange of information. The privileged, or insider process, sends information indirectly through shared resources to the unprivileged, or spying process. This can happen on different layers which have different transmission bit rates: OS (Xu et al., 2011), CPU cache (Ristenpart et al., 2009), virtual memory (pages) (Irazoqui et al., 2015)and network packets among others (Wu et al., 2015). Security such as a firewall, or an IDS (Intrusion Detection System) will not pick up this activity because it is not visible to access control mechanisms. Covert channel attacks are limited by the low bit rates involved and the cost of setting them up.

In 2005, Percival (2005) studied covert channel attacks in 2005 in both L1 and L2 cache with bit rates of 200kps and 100kps, respectively in a native system. His research led him to the conclusion that a covert channel attack on L2 cache was more practicable than on L1 cache because process core migration makes L1 cache more noisy. In 2006, Wang and Lee (2006) showed how to build a covert channel SMT/FU in which the attacker exploited OS vulnerability, interfering with target progress is through the shared pool of Functional Unit FU to interfere with the victim's process.

In 2009, following the rise of cloud computing, Ristenpart et al. (2009) demonstrated a functional high level covert channel attack against L2 cache with the low bit rate of 0.2bps, while 2012 saw research by Xu et al. (2011) into exploiting the memory bus in order to increase bit rate to 3bps, Maurice et al. (2015b) in 2015 demonstrated that limiting bit rate between two processes formed a barrier to covert channel attacks through the uncertainty created by a higher frequency of core migration. That study used inclusive cache L3 to overcome the issue and found higher bit rates than in previous attacks, with 751bps on unchanged attack settings.

2.5 Related Work

Side channel attacks have been studied in the laboratory and in real systems over the past twenty years. They have been practised against on on-board resources such as CPU computational units, cache and main memories. The majority of these attacks have been concerned with cloud systems, with an emphasis on IaaS (Infrastructure as a Service), in which the physical resources of the same machine are logically isolated across VMs. As the cloud grows in popularity, cloud providers need a full understanding of how such attacks threaten their privacy. Consequently, it is crucial that cloud providers and software companies take cognisance of what resources they have that are primarily used in cloud systems that are vulnerable to side channel attacks. In the previous sections, the past and ongoing side channel attacks against vulnerable hardware and/or software have been outlined. The following sections will continue the review on the detection and prevention systems used to mitigate such attacks. Finally, the limitations of the recent studies related to the proposed framework, as demonstrated in Chapter 4, have been summarised and linked to the coming chapters.

2.5.1 Mitigation Techniques

2.5.1.1 OS level

Recent studies have shown that Operating systems are vulnerable to side channel attacks. CPU designers as well as software companies have responded with more efficient hardware designs and data fetching mechanisms in order to alleviate the attacks on sensitive data. This section will describe the proposed mitigation techniques to support OS against side channel attacks and to note their drawbacks with consideration to the performance overheads of the system.

Earlier research has demonstrated the achievement of high resolution (Yarom and Falkner, 2014) and very fast (Irazoqui et al., 2015) side channel attacks through a Flush+Reload attack, which has the potential to exploit the systems characteristic **page sharing**, which is described in detail in section 3.2.1. These attacks leverage the shared pages, which are utilised by OSes to merge identical pages and to share them across online and concurrent processes (or VMs). The content is a shared library of the same applications on the machine, such as an AES shared library in relation to SSL implementation in Linux; the possible attacks against this setting are detailed in section 2.4.2.5. In a drive, free space can be leased to more tenants. In the early stages of the cloud, the cloud providers aimed to reclaim the maximum possible amount of memory by utilising shared pages. However, the disclosure of the page sharing

vulnerability has caused software industries and cloud providers to disable the page sharing feature, which was previously the systems' default setting. An example software product is VMWare, and an example cloud provider is Amazon EC2 (Maurice et al., 2015b; Zhou et al., 2016). To mitigate side channel attacks against page sharing, Maurice et al. (2015b); Zhou et al. (2016) proposed the kernel space solution CACHEBAR to provide concrete protection to shared pages across VMs in PaaS. The drawback to this proposal is the OS modifications and performance impairment, particularly in cloud systems.

Besides, according to the memory page sharing vulnerabilities, Irazoqui et al. (2015) showed that the usage of S$A against the LLC cache in large page settings exploited the memory page's vulnerability and enabled the AES secret keys to be extracted. In their study, the attack is able to resolve the memory addresses. This is possible because in large size page addressing, the attackers can see enough physical memory addresses to be able to identify the victim's most recent accessed cache addresses. The authors showed that the attack could be defeated if large size pages were disabled and if private cache slices were supported for each VM, thus preventing cache inferences by one VM on another. While this technique would prevent the attacker from deducing the cache slices that the target used, the number of large pages that outperform small pages file sizes is large. Large pages, on the other hand, can be compromised using a RowHammer attack as Pessl et al. (2016) suggested.

In order to take advantage of page sharing to resist potential side channel attacks, Kim et al. (2012) the proposed STEALTHMEM detection system is a technique that uses page sharing without opening the system up to side channel attacks. STEALTHMEM protects the data from cache-based side channel attacks using the page colouring technique. STEALTHMEM blocks the cache lines that hold sensitive data and prevents the eviction of the data by a side channel attack. The attackers' observations are degraded to the point where the attacker finds it impossible to deduce which targeted cache lines have recently been accessed by the victim. STEALTHMEM requires that the host OS keeps a note of where the sensitive data is assigned, and this is used for both data encryption and decryption at all levels of the memory hierarchy, from the cache through to the system's main memory. The price of this defence method is an overhead that unfairly penalises non-sensitive programs, but Kayaalp et al. (2017) has shown how to improve cache colouring in LLC so then the sensitive data is protected without STEALTHMEM and similar software modifications being needed in the operating system. These techniques, however, are expensive to implement. Another useful technique against side channel attacks is noise generation, which makes the attacker unsure what his/her observations mean. Varadarajan et al. (2014) suggested a noise generation technique for use in L1I and L1D caches.

Despite its optimisation features, **Out-of-order** 3.2.1 may cause degradation in the timing resolution to harden the side channel, but the attack techniques seen recently could use serialisation to overcome Out-of-order. Yarom and Falkner (2014) recommends the use of the instructions `mfence` and `lfence` to counteract this attack. Out-of-order has, however, counteracted a Meltdown attack. Meltdown is regarded as the highest risk side channel attack for its ability to compromise billions of user devices from mainframes to mobiles and across common platforms like Microsoft Windows, Linux and OSX by way of attacks through cloud providers.

Kernel Address Space Layout Randomisation (KASLR) protects the data by randomising the memory addresses so then the attackers are unable to deduce the targets' use of memory addresses. It is a technique that has been used in a number of ways, but Hund et al. (2013) demonstrated a generic attack on both AMD and Intel processors in both native and cloud settings against enabled ASLR in the host Linux and Microsoft Windows OSes. The study identified a region between the kernel and user space where candidates might lie using the vulnerability of the double page fault method which relies on physical addresses. Evtyushkin et al. (2016) exploited BTB (branch target buffer) contentions to mount a side channel attack against both user-space ASLR and KASLR. Gruss et al. (2017b) demonstrated a successful RowHammer attack using a side channel attack (Gruss et al., 2016a) against KASLR. Gruss et al. (2016a) enhanced the OS protection through KSALR to prevent the leaking of address information from the host's OS address translation layer by an unprivileged local attacker. The method was to isolate the address space between the system processes that manage and translate memory addresses and the user space processes. Gruss et al. (2017a) suggested using KAISER to prevent the hardware from leaking information from the kernel to the user programs by creating an isolation layer between the kernel and the user space.

Another way that operating systems are vulnerable is through keystroke stealing, in which an attacker constructs the keys that the victim has pressed through the exploitation of the kernel keystrokes when they interrupt the handler. Lipp et al. (2017) demonstrated that the keys pressed by the target could be stolen through a web application. KeyDown was suggested by Schwarz et al. (2017b), which fakes keystrokes, as a method of protection against Flush+Reload and Prime+Probe.

Cleemput et al. (2012) suggested that compilers could be used against side channel attacks if the execution time were made to be uniform by transforming the code in the AES algorithm. This study, however, also addressed its own limitations which were the hardware requirements, the code's complexity, portability and performance. Crane et al. (2015) suggested that injecting noise into the program upon execution could be used to

achieve control-flow diversity. This study provided solutions to the limitations discussed by Cleemput et al. (2012), but their solutions are specific to the application concerned, cannot be generalised and can degrade system performance as well.

SGX (see 2.5.1.3) is a method of prevention that works through the hardware, to which an element that is sensitive to data has been added. It was not long after SGX was proposed that several lines of attack were demonstrated. SGX-based protection is expensive to execute, generates an overhead cost and any attackers can evade it. Brasser et al. (2017) demonstrated a side channel attack which could extract data in SGX, even when the Sanctum protection mechanism is present.

Countermeasures proposed by other researchers include Costan et al. (2016)s proposal for a Sanctum protection model that flushes the L1 cache while the host OS performs context switching. Zhang and Reiter (2013) suggested that a Prime+Probe attack could be defeated by flushing L1D/I in order to avoid data dependency. Page (2003) suggested the addition of memory noise to the memory accesses in order to leave the attacker confused.

2.5.1.2 Application level

The application layer falls on top of the OS layer, in which any applications must access hardware resources through OS. However, there are strong protection mechanisms in the OS but unintentional improper software implementations can lead to security holes in the systems. Attackers can exploit these holes in order to achieve leakage attacks. Consequently, the researchers demonstrated how improper coding leads to the occurrence of side channel attacks. This section describes the existing techniques, as the programming guidance, used to protect the data against side channel attacks, particularly by making it harder for any attackers to guess secret inputs and execution times, which is the main source that side channel attackers utilise.

Side channel attacks have increased over the past ten years because of the weaknesses found in implementing cryptographic algorithms (Bernstein, 2005). Cryptographic software such as OpenSSL makes shared libraries available at the same time to multiple programs in order to save memory space. It allows for the software libraries to be updated, but this exposes the software to potential threats. Section (2.4.2.6) shows the library sharing mechanism in which the look-up table is at risk of being leaked by the attackers. Protection techniques have been developed at the application level to mitigate attacks of this sort.

Constant-Time is a form of cryptographic protection against timing-based side channel attacks. It is application-based. Constant-Time disrupts variations in memory access through secret elements in the cryptographic algorithms such that the attackers are unable to synchro-

nise with the victim's data, making it difficult for an attacker to seek out the target programs' secret elements.

Timing-based side channel attacks work by observing variations in the execution time of the cryptographic algorithms' secret elements. To combat this, techniques have been developed to make the execution times constant in both symmetric algorithms like AES and in asymmetric algorithms like RSA (Pornin, 2016). C language is used to implement sensitive libraries such as OpenSSL due to its fast access memory. However, Cauligi et al. (2017) addressed the C language limitation and concluded that it is unsafe to implement such sensitive programs. The study investigated the usage of bit-wise instead of `IF` statement (Acıiçmez et al., 2007; Osvik et al., 2006) due to the fact that the use of an `IF` statement in such programs causes branching. This makes it possible to measure the target programs' execution time. The study went on to make technical recommendations to cryptographic library developers. Bernstein (2005) demonstrated side channel attack protection through a constant-time solution, while Reparaz et al. (2017) suggested the Dudect tool to assess the tendency to leak information on the cryptographic algorithms and to evaluate between constant-time slicing (Käsper and Schwabe, 2009) and vector permutation by Hamburg (2009). The study compared the existing countermeasures with the AES T-table in an OpenSSL implementation. Implementing such solutions is difficult, especially in embedded systems. Coppens et al. (2009) suggested that time variations could be masked through the use of programming techniques that would eliminate the data-dependent control flow. Shinde et al. (2016) proposed the software-based defence of deterministic multiplexing in CPUs supported by SGX. Implementing ECDH encryption with Curve25519 in Libgcrypt requires the Montgomery Ladder algorithm together with a branchless formula if side channel attacks through high-level secret-input-dependent branches and memory accesses are to be avoided. A variety of detection mechanisms exist, and multiple researchers have suggested the use of hashing techniques to confirm a runtime programs' integrity in order to maintain the security of program execution flow in an untrusted environment. Kirovski et al. (2002) combined the hash function with cryptography, but the side channel attacks got beneath the integrity mechanisms to go on to threaten the target security.

Bruinderink et al. (2016) proposed a technique against BLISS using 3500 samples, but their attack only works if certain assumptions are met and if they failed to attack BLISS-b Ducas (2014). Pessl et al. (2017), on the other hand, proposed an attack that can attack BLISS and BLISS-b together, but it requires further actions to take place and needs 6,000 samples to mount a successful side channel attack

While solutions capable of protecting sensitive data have been suggested, they have

failed in practice due to the impracticability of implementing constant-time high-speed AES algorithms. Yarom et al. (2017) suggested CacheBleed as a way of eliminating noise, and this is generated in constant-time, through the use of conflicts in the cache bank to expose secret inputs and execution time differences. Genkin et al. (2017) carried this observation out down to the Libgcryptlibrarys' low-level side channel vulnerabilities in order to bypass the order-4 elements in the decryption routine in order to highlight the existence of the key-dependent vulnerabilities.

2.5.1.3 Hardware level

A number of studies carried out over the past ten years have examined many hardware-based vulnerabilities that have permitted side channel attacks. Most often, the hardware resource targeted by the attackers are CPU caches. Microprocessor designers have made physical changes to reduce the impact of such attacks. A study by Percival (Percival, 2005) into L1 cache data leakage attacks suggested that microprocessor manufacturers should disable cache sharing between threads and the core to prevent any data from being evicted from the cache lines. However, by disabling cache sharing across concurrent programs, this leads to degrading the system performance significantly.

Modern processors support the use of hardware configuration in relation to enabling and disabling hardware settings such as multi-threading. One of the objectives of multi-threading is to support the synchronisation between threads from the same core, and this is a characteristic that is central to a number of side channel attacks as seen in the past. Disabling multithreading would seem to prevent the attack (Kim et al., 2012). SEALTHMEM was proposed in the aforementioned study as an alternative way of mitigating attacks, but Zhang et al. (2012) demonstrated the possibility that the exploitation of L1 caches can support a side channel attack. They showed that implementing such an attack was not straightforward, but that it was still possible.

AES has received attention from researchers and microprocessor designers since Orange Book chose it as an effective data protection method. Intel (Gueron, 2008) announced a new instruction set, AES-NI, in 2008. The instruction set is executed by the processor using the AES-dedicated hardware to accelerate performance and security (Xu, 2010), but not all platforms and software libraries (including OpenSSL) have implemented this solution. This means that the vulnerability is still there for the side channel attacks to use.

Cloud providers plan changes in the cloud system infrastructure. Regarding the presentation of a number of side channel attacks, (Ristenpart et al., 2009)(Xu et al., 2011)(Wu et al., 2012) preceded with a demonstration that showed that constructing a communication

channel to separate VMs in Amazon EC2 was feasible. The demonstration included internal hardware resource maps. The provision of dedicated instances enabled Amazon to improve its service to cloud consumers, with an emphasis on the firm isolation of VMs from one another. The physical resources assigned to one tenants' VM are not shared with the other VMs. Since 2014, (Irazoqui et al., 2015; Maurice et al., 2015b; Yarom and Falkner, 2014) have targeted the Intel CPUs inclusive feature to create a bridge for data exchange between processes that are not related to each other. [21] was unsuccessful in an attempt to implement an attack on AMD CPUs thanks to differences in the processes' inclusive behaviour. It can be difficult to produce a physical change in a microprocessor because of the influence that this can have on the performance of the CPU. The negative impact of this solution on the performance of the system makes it commercially non-viable.

The majority of recent attacks have targeted the hardware vulnerabilities (of Technology, 2018,?), including the L1, L2 and LLC caches, but suggested solutions which could detect and prevent such attacks have been proposed in both hardware and software forms. Liu et al. (2016) suggested models for protection on LLC, which would protect the operating system layer through the use of the performance optimisation characteristics shared by each of a CPUs cores. They then used Intels' recently announced cache technology, Cache Allocation Technology (CAT), to give OS level protection against side channel attacks to the LLC. After that, they introduced CATalyst, which combines software and hardware support in the isolation of LLC slices. It binds each slice to a single core, preventing cross-interference by cross-cores.

Intel has given its modern processors extended SGX to combat side channel attacks, to reduce OS vulnerability and to prevent extraction by side channel attack on the secret keys from the cryptographic libraries. However, in SGX, there are no specific protections at the level of the architecture. However, Moghimi et al. (2017) suggested that CacheZoom is capable of extracting all of the AES' secret keys once it has obtained a reasonable number of samples. Then, Costan et al. (2016) demonstrated the Sanctum mechanism as a way of generating noise in the L1 cache memory as the target process carries out an enclave/non-enclave switch. Good as it was, this mechanism could not prevent Brasser et al. (2017), who showed that there was no interruption needed in the enclave in order to probe the cache, so then nothing occurs to trigger a mode switch or a flushing of the cache.

Researchers have suggested to CPU designers the possibility of adding an extra bit which could be used to signal system events or permissions. As well as software protection, Shinde et al. (2016) suggested that the addition of an extra bit would allow the host operating system to be notified of intended page faults so then they could avoid context switching. This is a

response to the way that attackers send instructions to the operating system to load critical information which is then encoded to give a definition of the systems' current state, including virtual and physical addresses.

2.5.2 Profiling-Based Detection Systems

Section 2.5.1 reviewed the proposed techniques and mechanisms in relation to the hardware and software used to prevent side channel attacks from stealing sensitive data. The fact is that these approaches and techniques have not, so far, been enough to secure sensitive data. Consequently, this section extends the study to include existing detection systems that are helping to detect and identify side channel attacks, particularly concerning detecting malicious VMs in cloud systems. Detection systems primarily rely on the observation of the attack activities by profiling the execution transactions of the side channel attack programs and taking advantage of the unintentional memory contentions that are generated by the attack programs.

Side channel attacks against the microprocessor components often go undetected because the attacks, which access the system control mechanisms, are hidden. Instead of accessing the targeted resources in a legitimate manner, the attackers exploit vulnerabilities in the hardware design and in the way that the software is implemented. Side channel attacks degrade system performance, and so recent research has shown that the best practice of acting against side channel attacks is by monitoring the overall computer performance, which is the main focus in this thesis. Researchers have used performance metrics as a way of detecting and identifying attacks. Detection of this sort can be classified as a source or an analysis approach.

A number of approaches have been used for side channel attack detection. Zhang et al. (2016a) proposed statistical analysis in order to identify cache attacks, relying on CPU cycles to monitor accessed and non-accessed cache (miss/hit) attacks. Briongos et al. (2016) proposed monitoring for Flush+Reload attacks against the AES algorithm and the study looked at the `clflush` instruction of multiple cache lines, which is the core of the attack. Detecting attacks in that case used the CPU cycle as its primary data collection source.

Section 3.2.1 discussed the power of `RDTSC` as a way of detecting differences in execution time of memory accesses to reveal which accesses were being used by the attackers with a very high resolution. As a form of detection, the CPU clock cycle has been used as a data vector, with which the researchers were able to separate the distinct patterns created by abnormal activities from the normal encryption processes. Briongos et al. (2016) used

the vectors of the observed CPU cycles to detect Flush+Reload attacks against AES using a statistical model to detect the cache line(s) flushed by an attacker while synchronising with the target to infer the targets look-up table. Their findings cannot be implemented in real-life systems because unexpected workloads could trigger false positives, thus reducing the accuracy of detection. The authors of the study suggested using machine learning to increase detection efficiency, but this would require more than only the `RDTSC` feature. HPC can provide a wide range of system events concerning the current state of program executions which can then be fed into a machine learning algorithm, yielding more efficient and accurate results.

Recent studies have showed that hardware-based malicious activity detection methods principally rely on High Performance Counters (HPC)s, in which special registers are built into mainstream processors. For more details, refer to section 4, on both data leakage and Malware studies. HPCs provide program execution granularity with a quantum precision. Table 2.2 shows some of the previous work which has utilised HPCs for both attack and defence in reference to both data leakage and Malware studies, unlike (Fei et al., 2014) which basically relies on `RDTSC` to collect the data in the analysis stage. Data HPC-based profiling has a larger capacity to capture the processors' state of various program execution characteristics as events. This allows for the machine learning method to be more feasible in the detection systems. Zhang et al. (2016a) proposed CloudRadar, which deploys a signature and anomaly detection system to detect side channel and memory Denial of Service (DoS) Attacks in cloud systems. HPCs also allow for machine learning algorithms to classify the attack pattern in the computational environment with efficient, reliable and highly accurate results. For instance, Alam et al. (2017) got 99% accuracy in real-time system monitoring. Zhang et al. (2012) successfully deployed Support Vector Machine (SVM) to support setup side channel attacks at the core level to establish an L1 channel between the attacker and the victim. SVM reduces the potential noise from different sources. Briongos et al. (2017) used machine learning to eliminate unstable program execution noise in relation to the synchronisation settings between the attacker and victim.

Vogl and Eckert (2012) proposed a trapping technique using PMCs to monitor programs at the instruction level inside VMs. The authors demonstrated the capability of the monitoring program's execution features by observing specific instructions inside the VMs in cloud systems. Kayaalp et al. (2017) suggested using Relaxed Inclusion Caches (RIC) to mitigate side channel attacks. Nomani and Szefer (2015) suggested a mechanism for detection and mitigation by injections (integrating or hooking) into the OS system scheduler to monitor how programs use memory as a way of detecting malicious programs. This study focused

Feild	Category	Publications
SCA	Attack	(Gruss et al., 2015b)(Irazoqui et al., 2015)(Gruss et al., 2017a)(Zhang et al., 2012)
	Defence	(Briongos et al., 2017)(Alam et al., 2017)
Malware	Attack	(Vogl and Eckert, 2012)
	Defence	(Wang and Karri, 2013)(Tang et al., 2014)(Demme et al., 2013)Pfoh et al. (2011)(Schwarz et al., 2017a)(Gupta, 2017)

Table 2.2 Categorise PMU-based attack and defence for side channel and Malware studies

primarily on CPU integers and floating point units and their influence on CPU component usage. It also used the machine learning algorithm's Neural Network (NN) to predict which applications would go on to be memory intensive. The fact is that NNs complexity in the training stage degrades system performance. Zhang et al. (2016a) designed CloudRadar as a protection system capable of detecting cross-VM side channel attacks on PaaS in public cloud services against LLC with no hardware or software configuration settings. Their model combines signature-based and anomaly-based detection methods and relies on the use of hardware performance counters.

2.5.3 Summary

None of the detection and prevention techniques developed so far can prevent a side channel attack from taking place. They can make attacks more difficult to carry out, but they cannot stop attackers from achieving their malicious aims. This is because vulnerabilities in the hardware and software allow the attackers to find ways around them. Ge et al. (2017) concluded that the most commonly used current processors, including x86 and ARM processors, are not designed to maintain security in microprocessors. Lipp et al. (2018) suggested to microprocessor manufacturers that performance should not be the sole aim of chip design. Instead, they should be more concerned with the security holes present when making hidden communication channels between two entities (threads, processes or VMs) in the computational environment to complete execution transactions securely.

What stands in the way of side channel attack mitigation is the overhead generated by the protection methodologies and the fact that implementing the methods is complex. The overheads in question relate to the fact that operating systems operate on one kernel space and that applications operate on user space. OS overheads negatively impact the system.

The fact is that the majority of both of the proposed side channel attacks and the countermeasures against them are reliant on assumptions. Enabling and disabling features is

an example. It follows that building a reliable security model requires the study of the vulnerabilities that attackers exploit and it notes the limitations in reference to the abilities of the existing countermeasures to provide security tailored to specific situations.

2.6 Limitations of Existing Works or Summary and Research Gaps

This chapter is a review of the recent work in order to reveal the most up-to-date best practices in side channel attack detection, addressing the limitations in the previous work on the subject. Commercial anti-virus software has had limited success in detecting side channel attacks. This is because the characteristics of such attacks are that they do not require privilege in order to succeed and utilise system calls. They instead rely on observing the effect of their own program executions on the same shared hardware and software resources. A side channel attack's primary aim is to discover vulnerabilities leading to the detection of data-dependency in secret elements. Computational noise in the CPU makes it difficult to use basic side channel attacks. Because researchers continually propose countermeasures when new forms of side channel attacks emerge, attackers are beginning to employ more than one technique to achieve their malicious intentions Kocher et al. (2018); Lipp et al. (2018).

In spite of the recent detection systems proposed to detect side channel attacks, they are liable in relation to one or more factors which indicates the efficiency and accuracy detection. In the following sub-section, the most recent relevant works regarding the proposed framework in this thesis have been listed according to the general detection framework components.

Profiling: Zhang et al. (2016a), Payer (2016) proposed the use of the `perf` tool against side channel attacks by monitoring existing processes in the system. Zhang et al. (2016a) selected VMs at random to monitor, while Payer (2016) monitored all processes in the system. The fact remains that side channel attackers can escape observation because the use of `perf` in both studies depends on the file system `proc` to retrieve information about the system's existing processes. A case study on Malware attacks Wang and Karri (2013) demonstrated the feasibility of an attacker modifying the `proc` file to hide its process id from the system so then `perf` gained no information about the malicious processes. In this paper, on the other hand, we have proposed a system profiling mechanism that targets the processor cores instead of the `proc` file system, so then all program execution transactions appear on the

observation. This means that the attackers cannot escape observation. This limitation is addressed in section 4.13

Native and Cloud systems: Alam et al. (2017) and Chiappetta et al. (2015) proposed detection systems using the machine learning approach to detect side channel attack in native systems. However, they failed to detect malicious VMs in cloud systems. Zhang et al. (2016a) setup a detection system in cloud settings, but their detection system worked only under certain conditions. For instance, the system requires dedicated hardware to be employed for the detection system and the signature of the cryptographic applications in VMs must be recorded. Fei et al. (2014) proposed a detection system in both native cloud systems, but the cost of VM detection was higher due to the extra analysis needed. Our work proposes a detection mechanism of *ML* in both native and cloud systems without an additional cost concerning the cloud systems.

This limitation is addressed in sections limitation is addressed in section 4.12 and limitation is addressed in section 4.13

Synchronisation: When the side channel attack uses hardware resources such as CPU cache memory, it basically relies on memory contentions in the repetition manner which leads to unintentional contentions. The attackers are unaware of this, and this causes significant abnormal activities. This can be easily detected by utilising a synchronisation approach. This means that the attack processes can be detected by relying on the data collected by the victim (Kulah et al., 2018). This approach is vulnerable in two potential circumstances. First, Allaf et al. (2017) studied a comparison of multiple machine learning algorithms, namely ageis SPEC cpu2006 int and fp application, in order to stress the CPU cache memory. As a result of this, heavy workloads have a negative impact on the detection accuracy of three machine learning algorithms including DT, PCAANN and k-NN. All algorithms performed well when no workload was running, with int applications such as gcc and bzips degrading the accuracy. With fp, the accuracy got worse.

Second, there might be smart side channel attacks deploy to evade the defence systems by slowing down the observations in order to produce irregular unintentional contentions by the attack programs. Allaf et al. (2018) proposed a detection mechanism that does not rely on the synchronisation approach. Instead, it monitors the scheduling quantum of each CPU cores in the system. The proposed detection requires a very low number of samples to detect and identify the attackers.

Previous works also showed that their detection system requires synchronisation. In this setting, both the victim and attacker programs are monitored to detect data dependencies while the attacker program is channelled with the victim programs to force the system to

trigger hardware contentions Alam et al. (2017); Briongos et al. (2016); Chiappetta et al. (2015); Payer (2016). The proposed detection systems are vulnerable for the reasons below.

Firstly, in the case where other normal programs use the same shared library as the victim, such as AES in the implementation of OpenSSL, the unintentional hardware contentions are feasible when multiple programs (or users) are accessing the same shared library at the same time. This leads to data dependencies. Thus, the normal programs are detected as an attacker.

Second, out of the workloads used in the test data-set, there is a negative impact on the detection accuracy (Allaf et al., 2017). Particularly, Gulmezoglu et al. (2017) proposed a detection algorithm to detect the attack by collecting L1 data. Intensive workloads degrade detection accuracy.

Third, in the case of smarter attacks, Gulmezoglu et al. (2017) stated that they can be evaded and the attack will be mis-classified. In the proposed framework, even if the attacker slows down the attack, the attacker can still be detected unless they slow down to a degree that cannot be beneficial in detecting any dependencies. This limitation is addressed in detail in section 4.12

Machine learning: In the previous works Alam et al. (2017); Allaf et al. (2017); Chiappetta et al. (2015), the utilisation of supervised machine learning algorithms needs to be trained for each of the cryptographic algorithms across existing attacks. However, Briongos et al. (2017) used unsupervised machine learning to detect side channel attack activities regardless of what techniques were used. The detection system incorrectly detected normal users, as mentioned in the previous section, who use the same cryptographic algorithm as the victim. Unlike the previous work, the proposed detection system uses supervised machine learning to detect side channel attack activities using the phase detection approach. The details have been given in section 4.3. This limitation is addressed in detail in section 4.12.

Performance: The performance overhead is the central issue affecting system performance in reference to any potential detection methods. System overheads need to be considered, and can be classified as either an OS-based overhead or an application-based overhead. An OS overhead is much more expensive than an application overhead. Nomani and Szefer (2015) recommends injecting a machine learning algorithm into OS scheduling to monitor CPU component usage to detect malicious processes. Cloudradar Zhang et al. (2016a) employs and dedicates three processor cores to monitor malicious processes. Payer Payer (2016) continuously monitors all existing processes. These mechanisms incur overheads in the host system. In the proposed framework, instead of injecting machine learning algorithms at the OS level, the detection system is placed in the user space and only data collection is placed in the kernel space. The proposed framework does not monitor the

whole processes in the system, and it instead profiles the processor core executions. Finally, the proposed framework does not require dedicated hardware and OS configurations. This limitation is addressed in detail in section 4.12.6.

Chapter 3

Preliminaries - Synchronous Trace-based Detection

3.1 Introduction

Data sensitivity is of increasing importance, and this is particularly true in cloud computing. The primary use of cryptographic techniques on the internet and in cloud systems is to protect sensitive data such as, among other types, patient records, banking transactions and social web accounts and posts. However, there have been consequent attacks designed to steal sensitive data that target critical cryptographic elements as secret keys, look-up tables and mathematical operations including square multiplication. AES, as an example, has been developed to protect data in a variety of domains including native and cloud systems. There has been a good deal of research into the use of side channel attacks on AES algorithms (Irazoqui et al., 2014).

As mentioned in Chapter 2, there are two main attack techniques, namely Flush+Reload and Prime+Probe, making the use of malicious activities performed during the attack stages. The attacks mainly target Last Level Cache (LLC). This is because LLC provides high bit rates to transfer the largest amounts of data as possible. However, there are different approaches that have been used in detecting such attacks such as statistical, probabilistic Fei et al. (2014) and machine learning approaches (Alam et al., 2017). For more detail, refer to section 2.5.

Fei et al. (2014) proposed a statistical analysis of the cache access driven attacks which rely on CPU cycles to monitor accessed and non-accessed cache-based (miss/hit) attacks. Briongos et al. (2016) suggested detection based on FR attacks against AES by analysing the flush instruction of the multiple cache lines which form the core of the attack. Their

model relied, for the most part, on the CPU cycle as a primary source for data collection. In another approach, the PMU registers were used to give greater granularity so then the features supporting the detection mechanisms could be extracted at a higher resolution. Zhang et al. (2016a) proposed CloudRadar, where the detection of signatures and anomalies were used to detect existing and new forms of side channel attacks as well as other cache attacks such as the denial of service attacks against CPU caches. The proposed framework requires dedicated resources to support detection. However, the proposed framework in this chapter does not require any additional resources.

It is the expectation that HPC provides more program execution features to investigate the attack activities. Therefore, machine learning algorithms can emerge to analyse complex data efficiently by detecting the attacker's malicious behaviour in the system. As the use of machine learning has been studied in a variety of domains, with particular emphasis on anti-virus work to protect individual computers, Intrusion Detection Systems (IDS) used to provide greater network security and spam detection to improve the security of information have emerged. Machine learning methods enable computers to build a data-driven model and to discover significant patterns of interest in the available data. We therefore have proposed a study to demonstrate the use of machine learning approach in detecting side channel attacks and how efficient is it in detecting side-channel attacks while they are synchronised with its victims. We then compared the three popular machine learning methods (NN; C4.5; and k-NN) to establish machine learning-based detection approach in order to demonstrate which will achieve the highest classification level in order to detect such attacks, followed by the impact of various workloads on detection accuracy.

3.2 Background

This section examines the previously suggested techniques and methods for both attacks and the detection of attacks, for which a number of approaches have been used. This section also examines the most important of the key components involved in attacks and in defence against attacks.

3.2.1 CPU Architecture and Components

The CPU, or central processing unit, is comprised of interconnected components that make it possible for programs to execute. For that reason, it is targeted by side channel attacks to a considerable extent. In a computational environment, its bus connections with on-

board resources like main memory and input/output (I/O) devices make it the main physical resource. This is because any data from any of the involved resources must be brought to the CPU for manipulation. Modern CPUs consists of multiple cores, which makes it possible to improve performance efficiency and to run a number of programs concurrently. Each core amounts to one logical isolated processor inside the CPU die. Each of which has its own components as follow:

1. **CPU Registers** The fastest memory in a computing system is the CPU register. It is also the smallest memory in the system. Registers are online memories and they are the place where any program instructions are actually executed. Registers are measured by the number of bits which they hold: eight; thirty-two and sixty-four.

   Registers fall into a number of types, each of which has the responsibility of a particular function. Some registers cannot be accessed by the programmers, only the OS. These include the Memory Address Register (MAR), the Instruction Register (IR) and the Memory Buffer Register (MBR), as well as the Temporary Register (TR). However, some registers are available for use by the programmers by utilising either an assembly language or another high level language such as C and C++. General Purpose Registers (GPR hold both program instructions and data. Debugging registers hold program states in real-time. Flag Registers (FR) hold information on the occurrences of particular conditions in the operations of the CPU. Debug Registers, Model Specific Registers (MSR) and Control Registers ensure the proper usage of the MSRs in capturing CPU events accurately. They are all used to improve system performance and to locate weaknesses in the programme. For security reasons, CPU manufactures, such as Intel and AMD, do not provide detailed documentation for these registers. However, researchers have found detailed information on the registers in order to propose their vulnerabilities (Kocher et al., 2018; Lipp et al., 2018). The focus in this study will be on MSR, which are charged with capturing events in which the CPU has an interest which occur in relation to the CPU components: cache misses, branch predictions and the total number of instructions executed. For more details on MSRs, see 3.2.3.2.

2. **Caches** are an on-chip buffer inside the CPU, and their purpose is to promote the efficiency of data transformation. In typical computer systems, memories are ordered by size (small to large) and by speed (high to slow). The registers include L1, L2, LLC caches and the main memory. CPU caches the buffer data that has been required from the main memory in a trade-off of performance against speed, so then the CPU has the data when it needs it. The buffer arrangement is sequential, with a typical CPU

comprising of a cache at three levels: L1, L2 and LLC, which maps the order from fast to slow and small to large (the larger it is, the slower it is) with the exception that L1 is split between instructions and data. Speed variations provide attacks, which run concurrently with the attacker's program, with their main source of information about the data prefetch activities of other programs (i.e. victims) which can be used for malicious purposes.

The caches are differently structured from level to level. The LLC cache is the outer level and is connected to the RAM directly. The basis of communication is through pages, with each page mapping the LLC cache sets and each cache set is divided into smaller units called **cache lines**[1] . Each cache line size varies from 4 to 64 bytes depending on the operating system settings. LLC provides the interface for L2 communication. The number of cache sets will generally be larger in LLC and larger in L2 than in L1. This is a hierarchical system and the data is tracked from RAM to L1. CPU caches are the essential intermediate between CPU computational units and the main memory. Each memory access is called a transaction through CPU caches. CPU caches are the resources that are targeted the most by attackers.

Cache memory is the most important determinant of performance in a computer system. A number of algorithms have been proposed to exploit these characteristics by which future data can be predicted. For programs with loops, algorithms exist to forecast which data will be needed next by the loop that is to be brought to the closest cache memory. Using these algorithms raises performance levels as the algorithm advises the operating system what data is about to be required. Thus, the CPU decreases the ideal state in which the CPU is waiting until the data is brought to the cache. However, side channel attackers have exploited these features to achieve their malicious intentions. Section 3.3 demonstrates how attackers can exploit these characteristics

3. **Miss/Hit events** are memory access occurrences that arise from hierarchical memory when data is buffered. Their main purpose is latency measurement to identify program bottlenecks. The CPU begins by asking for data from the lowest level (L1) and if the data is not found there, then it moves higher level by level until it reaches the main memory. Each request to move up one level is counted as a miss event at that level; when no miss event occurs, a hit is registered. Misses and hits can be gathered directly from the Hardware Performance Counters (HPCs) or by encoding the differences in time execution for the memory accesses, which is mostly used by side channel attacks.

[1] It is the smallest unit that can be dealt among cache levels to exchange data

Note, however, that miss/hit events have a greater complexity than simply being numbers. They can arise from a variety of sources. As an example, when the value of a variable is accessed in the main memory for the first time, there will be a series of misses as each level fails to provide the required value. There will be cases where the value has been temporarily evicted from the memory level that it previously occupied. This is dependent on the design of the hardware and on the programme, and attackers use such evictions in the measurement of data-dependent accesses.

The reason behind the different ways of organising caches is to yield the highest hit rate. This to avoid cache line contentions. However, such attacks take advantage of the contentions in order to encode the victims activities from the usage of the shared hardware or resources.

4. **Performance Monitoring Unit (PMU):** Programmers need help in debugging and locating the bottlenecks in their programs and PMU provides this assistance with run-time feedback. PMU offers services to programmers by detailing the current state of the internal CPU and its connection with I/O devices in real-time. More details are given in section 3.2.3.

5. **Timing** Timing program execution with a high resolution is a key side channel attack measure. Different resources exist to align the timing with the platform and CPU architecture. The timing can be used by taking the start time and end time of the motherboard's clock such as `gettodaytime()`, as in Linux systems. However, PMC, which sets one of the registers to count the CPU cycles, provides more details as given in the settings in Section 3.2.3.1. Time Stamp Counters (TSC) can also be used at a low level to count the cycle spent during execution of a program or a piece of code which is preferred by side channel studies due to its accuracy.

   Time Stamp Counter (TSC) is a 64-bit register that has been installed on every x86 processor since the Pentium. `TSC` observations are widely used in performance and side channel attack domains. It provides an automatic count of the number of cycles from the time which it is started to the time which it is stopped. Executing the `RDTSC` instruction accesses `TSC`, returning the `TSC` register contents to `EDX:EAX`. The main use of `RDTSC` is modelling a data leakage attack, especially in the observation stages such as Prime+Prob (Maurice et al., 2015b), Flush+Reload (Yarom and Falkner, 2014) and Evict+Time (Tromer et al., 2010).

6. **Out-of-Order and Serialisation** is a program execution mechanism used in multi-

core systems to execute instructions simultaneously that are subject to the availability of data and execution unites in a fair manner to avoid CPU stalls. Out-of-order does not guarantee the sequence in which the program instructions are executed. With an Out-of-Order execution, it is hard to accurately measure the data access time for a particular piece of code or instruction due to the interference of the counting cycles of previous instructions. Instead, it may count the operations of the different codes. Intel researchers Paoloni (2010) therefore proposed a solution to guarantee the precise measurement of the data accesses of a piece of code by serialising the `RDTSC` using `CPUID` with some operating system function support to pin down the `TSM` register for assignment to the selected code and to guarantee that the cycles to that code are counted. Furthermore, Yarom and Falkner (2014) proposed a different serialisation approach using `mfence` and `lfence` to accurately observe the cache activities with minimum costs.

3.2.2 Performance Measurement Tools

HPC are a set of registers in PMU used for performance measurements. The main use of HPCs is to measure the performance of a piece of code, program or system (Eyerman et al., 2006) and to find out the impact on the system's bottlenecks. The HPCs utilisation requires careful attention due to the complexity and permissions of its registers. There are many profiling tools and libraries used to abstract the complexity and to easily produce monitoring reports about the system and program activities in both user and kernel spaces. The following sections lists the most two popular tools which are used side channel attacks studies (Chiappetta et al., 2015; Nomani and Szefer, 2015).

1. **PAPI** In 1999, (13) introduced a portable library Performance Application Programming Interface[2] used to abstract the complexity of HPC utilisation (Dongarra et al., 2001; Terpstra et al., 2010). PAPI is a widely-used research tool, especially in high performance computing, to measure access time across the memory hierarchy and the usage of CPU components (Eijkhout, 2015). PAPI can monitor preset events when two or more events occur simultaneously, but large sets of events generate overheads. Johnson et al. (2012) extended PAPI-V in 2012 to provide support for VM. However, PAPI-V cannot support all events as the native does.

[2]is an opensource library, which is the specification of a cross-platform interface related to hardware performance counters in modern microprocessors including Intel and AMD processors. http://icl.cs.utk.edu/papi/

2. **Perf** Perf is a profiler tool made for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple commandline interface. Perf is based on the perf events system call interface for most of the involved tasks.

3.2.3 High Performance Counters (HPC)

This section reviews the HPCs built into modern Intel processors used for debugging and performance measurements. High Performance Counters (HPC) are a set of registers in Performance Monitoring Units (PMU). HPC gives the programmers runtime feedback to help in debugging and to help find software bottlenecks in critical parts of various programs. On the other hand, recent studies showed that the use of HPCs in the security domain have become popular, particularly in relation to malware and side channel attack detection. HPC consists of various sets of integrated registers used to set up very precise metrics for different measurement scales, ranging from micro-operations, pieces of code and applications through to the entire system. In a modern multi-core processor, each processor core has one HPC charged with capturing Off-core activities. Furthermore, a set of PMUs on the same machine are able of working together to monitor Uncore activities[3].

This study examines the PMU components used for settings, configurations and profiling for performance measurements. The main implementation of this thesis essentially relies on the utilisation of PMU in the proposed detection system. Consequently, the utilised components in this thesis have been detailed in the following sub-sections.

3.2.3.1 Events:

Event are the essential characteristics used to establish the metrics that are in turn used to measure performance, which is described as the occurrence of a hardware action in response to the usage of CPU components. Examples include the L1, L2 and LCC cache accesses and branch predictions. Events can be per core (Offcore) or per socket (Uncore). The focus in this thesis is only on Offcore events.

Supported events vary by manufacturer and processor model. Each CPU architecture comes with its own list of events. As PMUs are processor specific, they thus support two event types: one of which is architectural events which can be found in the CPUs which have the same physical components. For instance, LLC cache misses can be found in any CPUs that have an LLC cache memory, and the supported architectural events listed by CPU model

[3]For more details, visit https://software.intel.com/en-us/forum

Code	Description
309H	counts the number of instructions which are executed
30AH	the unhalted cycles of the processor core
30BH	the number of reference cycles at the Time Stamp Counter (TSC) rate when the core is not in a halt state

Table 3.1 Fixed function events

has been shown in Chapter 18 (Intel, 2014). Non-architectural events vary by processor model. Modern CPUs typically support hundreds of non-architectural events, which have been listed by CPU model in Chapter 19 Intel (2014).

3.2.3.2 Model Specific Registers

A Model Specific Registers (MSR) is a set of registers called Performance Monitor Counters (PMC)s. A PMC can record the fine details of low-level activities during program execution. The number of registers varies from one CPU model to another. There are two types of PMC Fixed Function, which are three fixed registers in a typical CPU used to count specific events as listed in Table 3.1 and General Purpose registers, which are model specific. They typically consisted of four registers used to count various events. The list of all supported events for each model has been listed in Chapter 18 (Intel, 2014). Unlike fixed function counters, they must be set before use. Figure 3.1 shows the layout of the PMC registers holding the counted value and the number of times that the specified event has occurred. Typical modern CPUs have seven 64-bit performance counters including both fixed and general purpose registers, only 48-bits of which are active, so the maximum value that the counter holds is 0xFFFFFF. If the counter exceeds this number, then it causes overflow errors. PMC registers can be set either to zero or to a selected value. In the performance tools, intervals are used to make the observation within a loop. With every iteration, the counters are reset. On the other hand, PMC registers set to a value to trigger overflow Vogl and Eckert (2012) are used to trap the attacker from using a system call. In this thesis, both fixed and general purpose counters have mainly been used in the data collection stage. In this chapter, all PMC registers have been used in the experiments equally. In chapter 4, the only general purpose registers used primarily contributed in the detection and identification phases.

3.2.3.3 Performance Event Select Registers

Performance Event Select Registers are a set of special registers that control and configure the PMC registers. The only registers that have been utilised in this thesis are listed as follows:

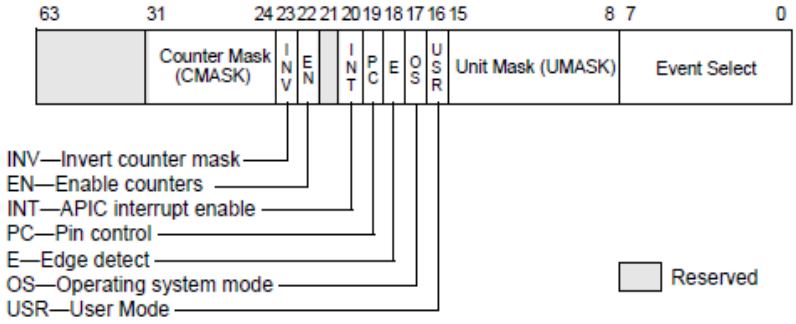


Fig. 3.1 Layout of IA32_**PerfEvtSel**$_{ith}$ MSRs

1. **PerfEvtSel***i*: the register responsible for setting the programmable counter registers. We can assume that a processor with four programmable counters PMC$_i$ registers is used to count four supported events by the processor, when $i \in \{1,2,3,4\}$. To operate the PMCi registers correctly in counting events, each PMC$_i$ register must be locally enabled by the **PerfEvtSel**$_i$ register and globally enabled by the overall register. Figure 3.1 shows the layout of **PerfEvtSel**$_i$ register.

   The first byte, which starts from bit 0-7, holds an event logic unit. The next byte, starting from bit 8-15, holds a unit mask. Any candidate event must comprise of a combination of these two. For instance, an event `UOPS_RETIRED.ALL` comprises `Umask:0x01 and event:0xC2 = 0x01C2`

   Bit 16 and 17 are used to determine whether the user space is excluded or included and the OS respectively, including kernel space activities, in profiling tasks. This option is contributed to in the profiling mechanism by removing the services which are working in the background.

   INT represents bit 20 in the register, enabling for an APIC interrupt to trigger an interrupt in the PMC overflow due to 0xFFFFFF being exceeded. The purpose of EN bit 22 is to enable or disable counting by the PMC register. However, in this study, overflow does not happen because the window interval for each sample is very small.

2. **IA32_FIXED_CTR_CTRL** The configurations of the fixed-function PMCs are done by writing to the bit fields in the MSR. The common operations in configuring PMC are enabling or disabling the event counters before and after specified tasks, and checking the status of the counter overflows, which is globally done by the following MSRs.

3. **IA32_PERF_GLOBAL_CTRL** This MSR can enable/disable the event counting of all or any combination of fixed-function PMCs and any general-purpose PMCs.

4. **IA32_PERF_GLOBAL_STATUS** This MSR allows for the querying of counter overflow conditions in any combination of fixed-function PMCs or general purpose PMCs.

5. **IA32_PERF_GLOBAL_OVF_CTRL** This MSR allows for the software to clear counter overflow conditions in any combination of fixed-function PMCs or general-purpose PMCs.

6. **Performance Monitor Interrupts (PMI):** A field in the control register that generates an exception through LAPIC in an overflow condition for fixed function counters and programmable counters PMCi. In Intel CPUs, IA32 PERF GLOBAL CTRL MSR provides single-bit controls to enable the counting of each performance counter. PMI fills in the various functions for a range of use-cases. It is used to detect faults when and where counter registers are improperly set. It is also used for the periodic event-based sampling of specified events on the PMCs. Finally, it has been used maliciously to detect system calls in Malware attacks (Vogl and Eckert, 2012).

3.2.3.4 Hardware Performance Counters Setup

The performance of measurement tasks requires setting and configuring the Performance Event Select Registers. Thus, this section illustrates the steps used to set up the PMC register to automatically count the specified events. Programming counters set the event-based interval and aspects of the profiling behaviour including the interval at which the samples should be generated.

Setting up hardware performance by user program or existing tools and libraries requires the `rdmsr` and `wrmsr` to read and write MSR registers, including PMCs required to count specific events. These instructions both use the `ECX` register to transmit the parameters for writing on the MSR registers or for getting the values of the PMC registers. The `rdmsr` instruction can also be used in the user space to read the fixed function[4] registers because they do not require a write on the MSR registers to specify an event.

Read operation: `rdmsr` instructions load the 64-bit contents of the specified MSR register into the `EDX:EAX` registers. The MSR registers the 32-bit high order as being loaded into the `EDX` register and the 32-bits low order into the `EAX` register. The `EDX:EAX`

[4]This does not work if fixed registers have been used by another service, unless the services are denied from accessing fixed registers such as `MNI` watchdog

combination generates the counters' actual number. This can be done in C using bit-wise operations.

```
((long long)EDX) | (((long long)EAX) << 32)
```

The **writing operation:** `wrmsr` cannot be executed in the user space and must be executed in the kernel space. Executions of the `wrmsr` instructions must therefore be in privilege ring-0. Therefore when PMC is used either by the user programs or by the tools, they must interface with the kernel as a driver to execute the `wrmsr` instructions .It then stores the contents of the `EDX:EAX` registers to a specified 64-bit MSR register. The contents of `EDX` go into the high-order 32-bits and the contents of `EAX` go into the low-order 32 bits.

Using `rdmsr` and `wrmsr` is not guaranteed to read or write the desired processor core PMC counters during profiling, especially if an interrupt is triggered. The use of Linux built-in interfaces is therefore recommended, and these are provided in `/x86/include/asm/msr.h` to set up an inter-processor interrupt. This will ensure that the MSR register read/write operations take place in the desired processor core. The interfaces are:

```
int rdmsr_safe_on_cpu(unsigned int cpu, u32 msr_no, u32 *l, u32 *h)

static inline int rdmsr_safe_on_cpu(unsigned int cpu, u32 msr_no, u32 *l, u32 *h)
```

Each of these functions takes on a single extra parameter - unsigned int cpu - which is the ID of the processor core used to guarantee that the profiling task will be pinned to the targeted processor core.

3.3 Threat Model

In this section, a brief microprocessor architecture design has been presented in relation to the side channel attack technique named Flush+Reload. The Malicious Loop (ML) is highlighted in a way that the detection system relies on.

In typical computer systems, there are four main memory layers to accommodate data when it is in a used state (DIU). They are hierarchically categorised from small in size to high in speed such as the L1 cache to large and slower, which is the main memory. Three layers (L1, L2 and L3 or LLC) are CPU caches used to buffer data in the main memory for a very short time. The data accesses are different at each layer; this is the key factor of side channel attack techniques based on access time or where they rely on data dependency. When CPU needs to access a piece of data in the main memory, it must be buffered in LLC, L2 and L1. In multi-tasking systems, an OS dedicates a region of memory for a program that is

logically isolated from the other regions of other programs in a manner so then the processes of one program cannot access the regions of other programs. This isolation is secured by the OS. However, in some OS settings, the OS shares a region of memory across multiple programs (i.e. shared library in Linux and DLL in MS Windows) to optimise performance and to reclaim more memory. Here, side channel attacks come into play by exploiting the resources and shared features.

In this model, a Flush+Reload attack technique is utilised to find the memory contentions while attacker and victim are using shared library. The AES algorithm is used to encrypt the plain text for both the attacker and victim. In Figure 3.2, the yellow page is the shared library in which the AES components are stored. The main components that are compromised by the attacker includes the look-up table (T), which is an array of secret elements which replaces the run-time computation with a simpler array indexing operation to generate a cipher text. Inside the victim, the shared library is utilised to encrypt the plain text. In the attacker's program, there is a ML used to scan T from the beginning address of T_0 to the end address of Tn to find out which element has recently been accessed by the victim, which is called data dependency. ML has two main tasks, one of which is to flush an address within the range of the addresses that the T_i is stored. We assume that this is used by the victim processes, and that this is followed by accessing the flushed address. This is the key feature in the proposed framework concerning the detection and identification tasks. In step one, the attacker flushes an address in T and waits for a very short time to observe the victim's access to the flushed address. When the victim accesses the flushed address, the data in the flushed address is brought to the LLC. In step three, the attacker accesses the flushed address and compares the access time. If the access time is approximated to the threshold, which is the cache access time excluded from the main memory access time, then this indicates that the flushed address has recently been accessed by the victim. This way, the attacker encodes the observations stored in an array, and then statistical analysis is applied on the off-line data to deduce the secret key.

3.4 Methodologies

This chapter presents three common supervised algorithms to classify Flush+Reload and Prime+Probe side channel attacks against AES, and then to compare the results of each method to determine which one most efficiently detected the attack under different workloads and the negative impact on accuracy.

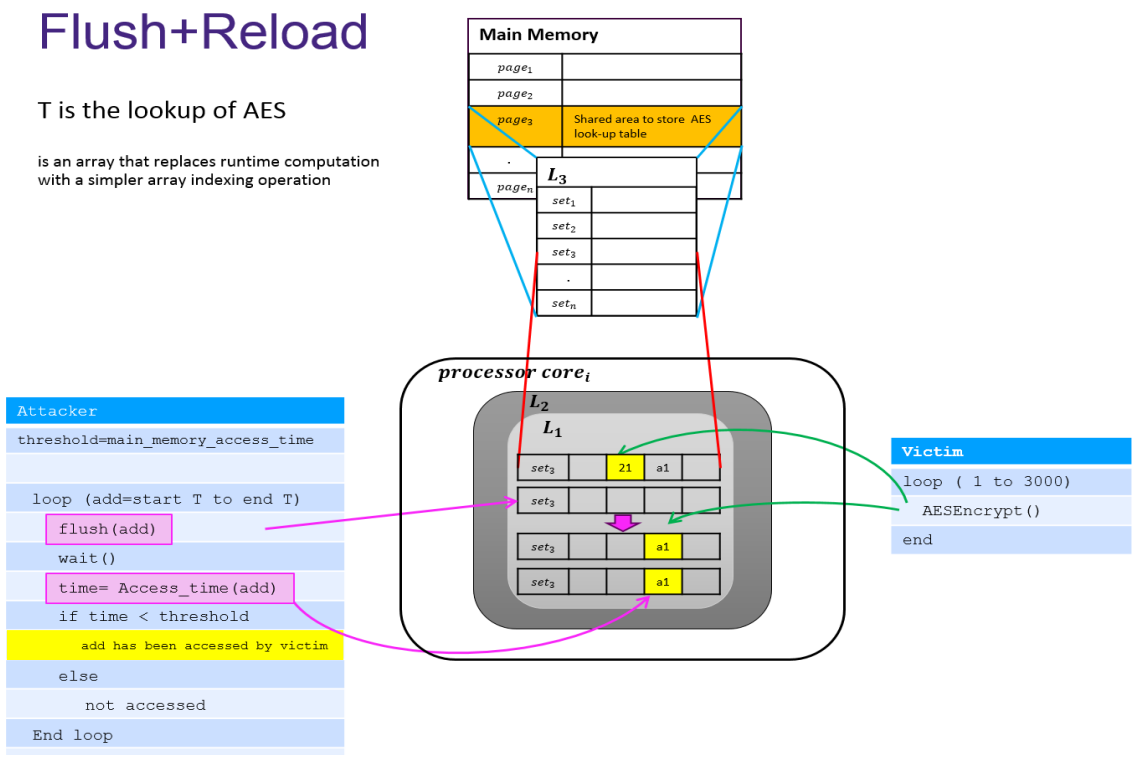


Fig. 3.2 A typical Flush+Reload attack against AES

3.4.0.1 Classification and regression or prediction

Most machine learning algorithms can complete classification and regression tasks. Classification allows for the prediction of exact classes from a given data-set. For example, each instance in the data-set must be either malicious or benign. Regression, on the other hand, predicts continuous values and not classes - it might predict prices, distances or weights items in a class that would be likely to be fetched, given the features that each possesses.

3.4.0.2 Bias and Variance

are key components in the accurate measurement of classification and regression tasks. Researchers use bias and variance to optimise the classifier (classification) or predictor (regression) models. Adjusting the degree of each bias or variance will affect the prediction. Building an optimal model requires a trade-off between bias and variance that will be arrived at based on the nature of the data (Breiman, 1996b).

1. **Variance or over-fitting:** An algorithm with very high variance pays excessive attention to the training data and fails to generalise a new model for the unseen data. It is basically memorising the training data instead of generalising. When it receives a new

data-set that does not closely resemble the training data-sets, it has no way of dealing with the unseen data.

2. **Bias or under-fitting:** High bias means that the training data contains errors; an algorithm with a very high bias pays little attention to the training data, and whatever actions the training data might be encouraging. The errors can be analysed in different ways, including low r2 and a large sum of squared error (SSE), among others.

3.4.1 Principal Component Analysis (PCA)

PCA is an unsupervised machine learning algorithm widely used in dimensional reduction to facilitate classification. It is a simple and widely-used algorithm which can be used to fin the direction of the spread of the data with the greatest variance before generating new coordinates.

3.4.2 Neural Network (NN)

NN is a supervised machine learning algorithm. It can be used to build a predictive model by learning from historical data and using the patterns to throughput binary or multiclass classifications.

The ability of NN to self-learn from examples allows the researchers to train NN with features from CPU events from which it acquires the knowledge to classify CPU activities into malicious and non-malicious respectively. Neural network architecture can generally be categorised into single-layer feed-forward networks, multi-layer feed-forward networks and recurrent networks. A number of other types have emerged including perceptron, backpropagation, self-organising map, adaptive resonance theory and radial basis function.

Ngiam et al. (2011) showed the efficiency of the algorithm in dealing with low dimensional data sets. This can work more efficiently with PCA, which reduces the dimension of the data. To accelerate the learning process, we used PCA to reduce the dimension and to then pass it on to the optimisation algorithm, L-DFGS, which is efficient for small data sets.

In choosing between the three activation functions, we have considered speed and accuracy. Because attacks are fast, data can be retrieved in less than one minute. Recently, Kingma and Ba (2014) introduced Adaptive Moment Estimation (ADAM), an optimisation technique used in NN which is fast, computationally efficient and requires less memory than DFGS. It also deals efficiently with large data sets. The Quasi-Newton method, on the other hand, is computationally expensive and requires more memory to store the Hessian matrix,

while LDFGS accelerates the speed of deep network learning (Dean et al., 2012). They used the algorithm for large data sets and showed it to be faster than the SGD algorithm. Limited-memory DFGS do not store Hk and are therefore faster than DFGS. Faced with a large data set, as we mentioned, ADAM is faster. Therefore, ADAM was used as the activate function.

3.4.3 K Nearest Neighbour (k-NN)

The instance-based algorithm k-NN is a simple non-parametric classification algorithm that is long-standing (Cover and Hart, 1967). It may be used in any classification task using discrete data but the classification of unseen data-sets and regression tasks to predict a continuous label relies on data-sets based on a time series. Each tested data class is predicted by measuring the test data items' similarity. The classification process was conducted on the test data-set realised for each class and its k closest neighbours. Any set of sample data points can be classified according to its neighbours' majority vote. k-NN makes use of a search engine based on the measurement distance functions to find the closest data items to the data-set. k-NN has been studied for a considerable period of time and a number of distance measures have been used, with the most popular being Euclidean, Hamming, Manhattan and Minkowski. This study has made use of the Hamming measure to find the best k instance for the classification tasks in the training data-set.

$$D_H = \sum_{i=1}^{k} |f - y| \tag{3.1}$$

Optimal k values are found on the basis that larger values mean a better classification. Since this approach is not reliable, this study uses the cross-validation (CV) Arlot et al. (2010); Kohavi et al. (1995) to determine how optimised the k value is. Cross-validation divides the data-sets into a number of predefined data-sets before feeding them independently to k-NN during training and testing tasks. The optimal k value is selected by the search engine from a number of independent predefined data-sets.

The k-NN algorithm measures the distances between the data items in the data-set. This is why k-NN has been chosen; the choice of data set rests on the data sample similarities with stress on the features that are near to their neighbours. The features chosen for this experiment include L1, L2 and LLC cache misses. This is because Flush+Reload attacks work by flushing a specific memory address from all levels of cache (L1, L2 and LLC) and, after a very short sleep, accessing the memory address that was flushed. Three consecutive cache accesses are needed; the memory access instruction generates an identical number of

hardware events, which in this case consists of cache misses, for each cache level. k-NN is looking for data items with the least distance between them, and so identifies efficiently the ML inside the Flush+Reload program.

3.4.4 Tree Algorithms

Tree algorithms work in a divide and conquer fashion by partitioning a given data-set recursively using either a depth-first Hunt et al. (1966) or breadth-first Shafer et al. (1996) approach until all of the data elements belong to a specific class. Tree algorithms attempt to detect the factors which affect the change occurring due to a specific event. It builds a tree structure-like model to efficiently predict unseen data (test data). Tree algorithms build classifiers in two phases. First, is building a tree for classification purposes. Second, is pruning the tree to generalise unseen data. The tree structure is composed of root, internal and leaf nodes. Root node represents the best feature of the data-set used to partition the data-set and the leaf nodes are the class labels. The internal nodes are generated by utilising impurity measures. The tree model is then transformed into a set of `if-else-then` decision rules and unseen samples are traversed from the root to leaf nodes to indicate one of the given class labels.

Impurity is the core function in tree algorithms and heavily affects the tree's performance. Impurity measures split the nodes of all available features and then selects the features which results in the most homogeneous sub-nodes. Impurity indicates the degree of homogeneity of the new sub-nodes to find out which sub-node is more homogeneous by checking all available features. The most common splitting criterion employed in impurity functions are as follows.

1. **Entropy (EN)** relies on information theory to measure which subsets require more information to indicate the degree of impurity in the same manner as if all elements in a subset are homogeneous. If the entropy is zero, then this leads to gain high accuracy. Whereas if the elements are equally distributed in a subset, then subset has an entropy of one and consequently, the model fails to efficiently classify the data. After calculating the entropy of each of the generated subsets, the sum will be compared with the entropy of the parent.

   Let's assume that D is a data-set which contains samples from c classes. The impurity functions for EN and GI are defined as follow:

$$EN(D) = \sum_{i=1}^{c} - p_i * \log_2(p_i) \tag{3.2}$$

where P_i is the proportion of class i in D.

2. **Gini Index (GI)** tries to minimise mis-classification by measuring the total variance across the classes. Where the Gini function returns zero, this is when the best separation can be achieved or return one when the distribution of the classes is 50/50. Consequently, the higher value of Gini function indicates higher homogeneity.

$$GI(D) = 1 - \sum_{i=1}^{c} (p_i)^2 \tag{3.3}$$

$$Gain(T,F) = IM(T) - IM(T,F) \tag{3.4}$$

When Impurity function $IM = EN = GI$, $0 \leqslant IM \leqslant 1$, $T =$ target variable and $F =$ Feature to be split on $IM(T,F) =$ the impurity is calculated after the data is split on feature F

$$IMGain: \ G(T,X) \ = \ \sum_{i=1}^{X} P_i * E_i \tag{3.5}$$

C4.5 is one of the supervised machine learning algorithms. It is uniquely easy to read and understand. The goal is to build a model that predicts the value of a target variable by asking multiple linear questions one by one to create a boundary. Future data is classified using a very simple data structure called a Tree. It is a statistical classifier like other classifiers, and uses a set of data to train and build a decision tree model using the concept of information entropy. The trained data is split into n-dimensional vectors which represent the features of the sample data and its class.

3.5 Model Evaluation Metrics

This section introduces the existing techniques which are used to evaluate predictive models, especially the classifiers generated from the supervised machine learning algorithms.

In supervised machine learning studies, there are various algorithms employed to solve prediction problems including classification and regression. For each, there are different metrics that have been utilised to measure a model's performance in predicting the classes.

When the machine learning algorithms are used for classification problems, a classifier model will be built to predict unseen data for the potential classes. For instance, in binary classification there are two classes of interest that the classifier should recognises them, in our case they are normal and attack classes. Then the model builds a set of rules for both classes, then unseen samples traverse through the model to output the result based on the defined matching the pattern. After the model classified unseen data, the model needs to be assessed its performance to test how well the model classifies unseen samples. Therefore, a number of metrics have been defined for assessing the classifiers' performance. Training sets are fed to the algorithms to build the classifiers. However, 100% correct prediction of unseen data cannot be guaranteed; there might be failure to predict some samples in the testing stage, especially in noisy data-sets. Thus, there are various techniques to address the prediction errors and based on that, further courses should be taken in the training stage such as data pre-processing to improve the model's performance and robustness.

In this thesis we have used three different methods to assess the machine learning algorithms' success in predicting the existence of a side channel attack. It then measured the overall accuracy to show the best candidate for the detection tasks. The next sections will discuss the most common metrics, which were used by the researcher in the context of the classification problems to visualise model performance and to indicate how far the predictions are from the actual values. Therefore, in the following subsections, more details about the evaluation measurements and their components will be given.

3.5.1 Confusion Matrix

The confusion matrix contains information about the predicted classes produced by the classifier models and the actual classes from the original data-sets. The confusion matrix fairly provides the finer details of the classifiers in predicting unseen data. The confusion matrix can be utilised for binary and multiple class classifications. A confusion matrix used in relation to a binary class classification is a $(2x2)$ matrix, whereas for multiple classes, there is a (cxc) matrix when c is the number of classes. Through the confusion matrix, most of the performance measurement metrics can be derived from the confusion matrix for various purposes. Table 3.2 represents the actual classes (presented in columns) against the predicted classes (presented in rows) in a $(2x2)$ matrix. The first row in the matrix shows the number of positive classes that the classifier predicted and the second row represents the negative classes. Before giving any details on the measurement metrics, there are terminologies in the confusion matrix which are described as follows:

	Prediction	
	Positive (P)	Negative (N)
Normal (P)	TP	FP
Attack (N)	TN	FN
	Total Positive	Total Negative

Table 3.2 Confusion Matrix

1. **True Positive (TP)** is the case where the classifier correctly recognises the positive samples in the data-set. For instance, if there are n positive samples in the actual class and if $TP = n$, then this means that the classifier 100% detected the positive classes.

2. **False Positive (FP)** in this case, represents when the classifier miss-classifies the positive classes as negative. FP =total number of actual positive classes $-TP$.

3. **True Negative (TN)** represents the total number of the negative classes detected by the classifier correctly.

4. **False Negative (FN)** when the classifier miss-classifies n samples of the Negative classes as Positive classes.

The ideal case for a classifier is when $FP = 0$ and $FN = 0$, which means that all potential classes are predicted correctly. $TP =$ total number of positive samples in the actual class and $TN =$ total number of negative samples of the actual classes, which means that all of the positive and negative classes are predicted as actual classes. However, in real world problems, it is not guaranteed that a predictor of 100% classifies the class of the target as the actual class where unseen samples are fed to the models. This leads to generating FP *and* FN in the model outcome. Therefore, most of the efforts are focused on minimising FP *and* FN. Minimising FP *or* FN relies on the business needs and the context of the problem that is going to be solved. In some cases, it is advised to minimise FN rather than FP, because FN is more important than FP or vice versa.

3.5.2 Evaluation Metrics

After building a prediction model, we need to make sure that the model is efficiently applied in the unseen data-set. Therefore, there are a number of metrics that can be used to assess the model. Different evaluation metrics are employed for different machine learning algorithms. For instance, for unsupervised machine learning algorithms, there are a set of metrics and for supervised algorithms, there are a different set of metrics. In this study, only classification algorithms have been used, thus the focus will be on the metrics that are commonly used in supervised machine learning for binary classification problems. As binary classification is used to classify between normal and attack activities in the system, we will describe the metrics that have been utilised in this thesis.

To explain the following metrics, let us assume that algorithm 1 captures 100 samples of real-time program execution activities; 5 samples are actually attack activities (Positive) and the rest are normal activities (Negative).

1. **Recall/Sensitivity (True positive Rate (TPR))** corresponds to the proportion of normal activities that are positive samples.

   Recall reveals what proportion of program activities actually were attacks, and what were classified by the algorithm as attack activities. The actual positive samples are the normal activities equal to the sum of TP and FN. The activities classified by the model that are normal are TP; refer to Table 3.2. Recall is more about capturing all of the samples that are attacks with the answer as attack. In equation 3.6, if the model recognises the actual 5 attack activities correctly, then the recall of the model is 100%.

$$Recall/Sensitivity\ (True\ Positive\ Rate) = \frac{TP}{TP+FN} \tag{3.6}$$

2. **Precision (True Positive Value)** measures what proportion of the program execution activities which are classified as attack activities are actually attack activities. The predictive attack activities (positive) are the sum of TP and FP and the actual attack activities are TP, as expressed in equation 6.

$$Precision\ (True\ Positive\ Value) = \frac{TP}{TP+FP} \tag{3.7}$$

3. **Specificity (False Positive Rate (FPR))** corresponds to the attack activity samples which are incorrectly classified as positive. Specificity exposes the finer details about what proportion of the program's execution activities are normal and what were

classified as normal program execution activities otherwise. The actual negative samples are equal to the sum of the FN and TN from the predictive model and the program execution activities as classified as being normal (TN).

$$Specificity(False\ Positive\ Rate\ ) = \frac{FP}{FP+TN} \tag{3.8}$$

4. **False Positive Value** reveals how many negative samples are correctly detected by the classifier; if the FPV is high, then it should be close to 100.

   This tells us how many of the test negatives are true negatives and if this number is high (should be close to 100), then it suggests that this new test is doing as good as the gold standard.

$$False\ Positive\ Value = \frac{FP}{FP+TN} \tag{3.9}$$

5. **Accuracy** is a metric used to indicate how the model is at predicting the samples correctly in the data-set. This also refers to the total number of correct prediction out of the total number of samples in the training data-sets. The ideal values is one Total $\frac{Total\ Numberof(TP\ and\ TN)}{Total\ Numberofsamples}$. Accuracy is recommended for balanced data-sets. The accuracy results are shown in the experiment results in this chapter.

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FN} \tag{3.10}$$

3.5.3 Receiver Operating Characteristic (ROC) curve

The ROC curve as proposed by Bradley (1997) is a graphical tool used for visualising predictive model performance metrics. It draws line graphs between recall or sensitivity and specificity. Points on the curve are the ratio between 0 and 1. The diagonal line from (0,0) to (1,1) indicates a random guess, which has 50% accuracy. Anything below this line is even less likely to be correct than a random guess, while anything above the line will range from good accuracy to excellent accuracy. The closer the line is to the top right corner, the better the performance.

3.5.4 Cross-Validation

The model needed the ability to generalise future data-sets. The data-set must therefore represent the problem that is to be solved. The data-set was divided into training and testing data-sets. First, the algorithms were trained on the training data-sets, after which they were evaluated against the test-set. The trained classifier models did not work on the test-set, so predictions on the testing set showed the general accuracy of the classifier. To ensure that the selected data-sets represented the problem requiring a solution, a technique called Cross-Validation (CV) was used. CV shuffled the data-set, including its attributes and labelled classes. It then separated the entire data-set into a large number of equal sized groups of instances, which are called folds. Each fold was treated as a new data-set, and each was divided by the user into the recommended percentages of training and testing sets; 80% training and 20% test is one possible division. The data-sets were then fed into the algorithms and their performance was monitored to evaluate their efficiency and accuracy. Cross-validation transforms the whole data-set into training and testing sets as an alternative to using separate testing and training sets. All of the data-set is involved in the transformation.

3.6 Synchronous Trace-based Detection

This section demonstrates the side channel attack detection using synchronisation approach.

3.6.1 Hardware and Software Specifications

The experiment was conducted on HP Proliant DL360 G7 with Intels Xeon X5650 2.66GHz processor with 16 GB RAM running Ubuntu 14.04. The various tests used SPEC CPU2006.

3.6.2 Experiment

We have conducted an experiment study by using the data collected from the experiment itself. We created an agent process that encrypted the fake data with the intention of simulating a victim, and used the custom Loadable Kernel Module (LKM) to access the PMC registers with minimum overhead in order to gain high resolution data. Our data set consisted of f features, when $F = \{F_1, F_2, F_3, ..., F_f\}$ and $f = 7$, because only seven PMC counters were available on the CPU, which is used in the experiment, including three fixed events (core cycles, reference cycles and core instructions) and four more efficient programmable events. For this experiment, we selected the most efficient events that had a positive impact on the

classification of the selected methods by considering their relationship to the attacks. In this experiment, we collected n samples, when a set of samples $S = \{S_1, S_2, S_3, ...S_n\}$, so then $n = 100$ and 50% of the samples are recorded while the victim is encrypting fake data and the other 50% of the samples are the samples when the victim is synchronised with the attacker while both are encrypting data and accessing the same shared library. Each sample row was arranged so then $S_i = \{X_1(1,v), ..., X_i(f,v), y_i\}$, when $1 \leqslant i \leqslant n$ and v is the number of encryption iterations executed by the victim. In this experiment, we used 3000 iterations for each event so $v = 3000$. Thus, X_i represents e executions of AES encryption function with the set of feature F. Each row samples S_i is labelled by y_i, which is the binary class that represents either an attack or normal situation. In this experiment, half of S are labelled as normal and the other half as attack activities.

The data was collected under two scenarios; one for light and one for heavy workloads. In the first scenario, high resolution data was secured by running only victim and attack programs. In the second scenario, we added noise by running additional applications from SPEC SPEC2006[5]; two int applications (`bzip2` and `gcc`) and two floating applications (`bwaves` and `dealII`)

The data-set was split into training and testing sets. The training sets contained 80 samples and 20 testing sets. To determine the influence of the different data set splits under each method, we split the data sets randomly into 20-fold cross validations. The training set is given to the machine learning algorithms to build the classifiers and the testing set is given to the classifier to evaluate their performance.

In this study, we have shown the impact of MLs running inside the FR and PP attack programs on the victim processes, which use a cryptographic algorithm to encrypt the sensitive data. Our hope was to detect the attack in both light and heavy workloads. The attacker would try to interfere with the victim processes and to synchronise itself with the shared LLC by monitoring its cache memory activities and using statistics to deduce the cache lines most recently used by the victim. We also hoped to detect the attack in the shortest possible time of less than 5 seconds; an efficient attack Irazoqui et al. (2014) requires over 50 seconds to recover all of the key fragments. This experiment can be applied in cloud systems, except for the additional overhead which is produced by an additional translation layer. This definitely reduces the resolution rate detection, as the most recent detection work Gulmezoglu et al. (2017)Briongos et al. (2016)Zhang et al. (2016a) shows the difference in accuracy rates between the native and cloud systems.

The shared library co-allocates two unrelated processes on LLC to the same machine.

[5]SPEC SPEC2006 is widely used to evaluate performance of computer systems https://www.spec.org/

Table 3.3 Classification Accuracy for the three methods C4.5, PCANN and k-NN, against two attacks Flush+Reload (FR) and Prime+Probe (PP).

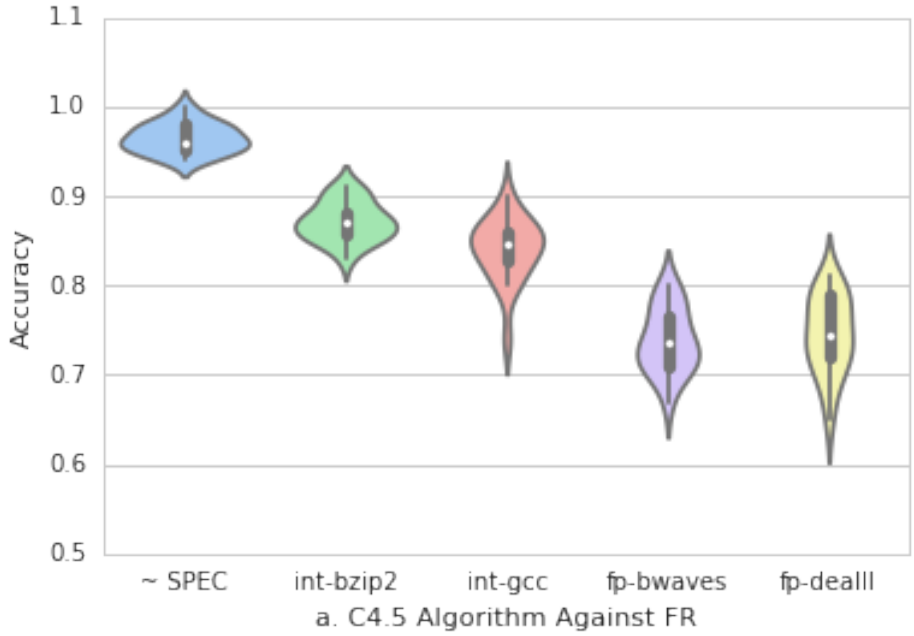
Classification Accuracy on FR and PP							
Type	Bench\Attack	C4.5		NN		k-NN	
		FR	PP	FR	PP	FR	PP
∼SPEC	No SPEC	0.97	0.98	0.93	0.76	0.85	0.83
SPECint	bzip2	0.91	0.96	0.8	0.8	0.8	0.78
	gcc	0.87	0.94	0.77	0.79	0.84	0.8
SPECfp	bwaves	0.74	0.74	0.73	0.73	0.73	0.74
	dealII	0.75	0.7	0.7	0.64	0.63	0.7

Thus, we are able ot detect malicious FR and PP attack activities when a ML is run to synchronise with the victim's processes in order to give the attacker a chance of accessing the shared memory. The aim of our hypothesis was to evaluate the best classification method and the impact of SPEC in detecting such attacks with a high rate of accuracy even with the loading of the benchmark.

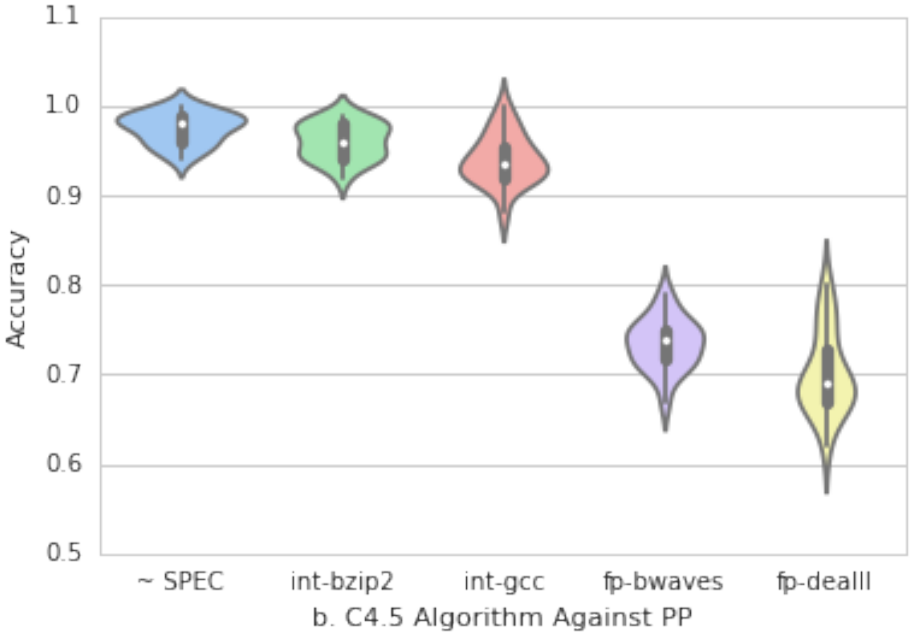
3.6.3 Result Analysis and Discussion

We are looking for an optimal classifier that works accurately and efficiently among the selected methods under both light and heavy workloads. Each of the three algorithms presented in the previous section was run on each of the 20-fold splits of the data set divided into training and testing sets respectively. Based on the previous studies and the results gained from our experiment, we compared the methods based on accuracy and efficiency because these two factors are important to the victims when dealing with sensitive data. For accuracy, the victim needs to correctly classify the attack. For efficiency, the victim needs to detect the attacks quickly before the attacker retrieves the whole key-bits and disrupt the attack.

The results, as presented in Table (1) and Figure (1), show the accuracy of the side channel attack classification including the FR and PP techniques for all methods in three scenarios without SPEC ($\sim$ *SPEC*), CPECint or SPECfp. The C4.5 algorithm performed with the highest accuracy in all scenarios in reference to detecting FR. Without SPEC, the success rate is 0.97%. This causes a fall in SPECint to 0.91% and to 0.87% in `bzip2` and `gcc` respectively. There is a further decrease from 0.74% in SPECfp and to 0.74% and 0.75% for `bwaves` and `dealII` respectively. However, in PP detecting, it classifies better in SPEC,



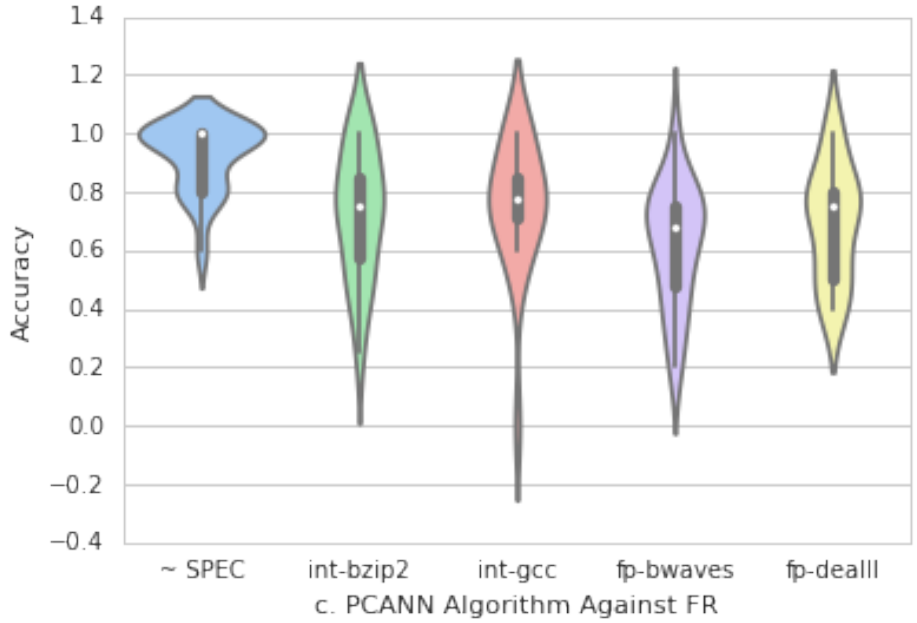
(a) C4.5 Algorithm against FR



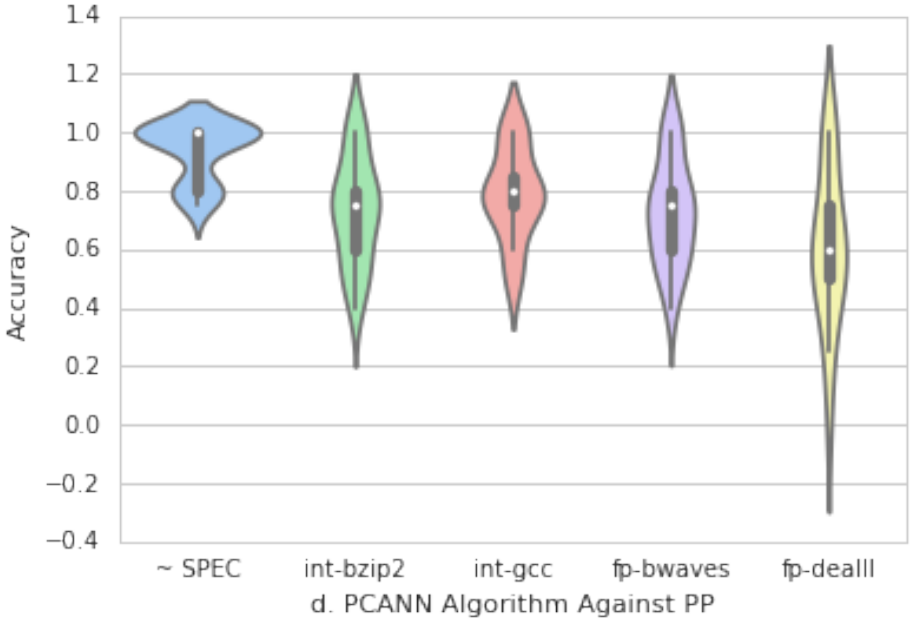
(b) C4.5 Algorithm against PP

`bzip2` and `gcc`. It stays the same in the `bwaves`, but it is worse in `dealII`. This is because, in a `PP` attack, the attacker uses more CPU components and this maximises the number of occurrences of specified events.

PCANN is good at detecting FR without SPECint and SPECfp, but it performs poorly in

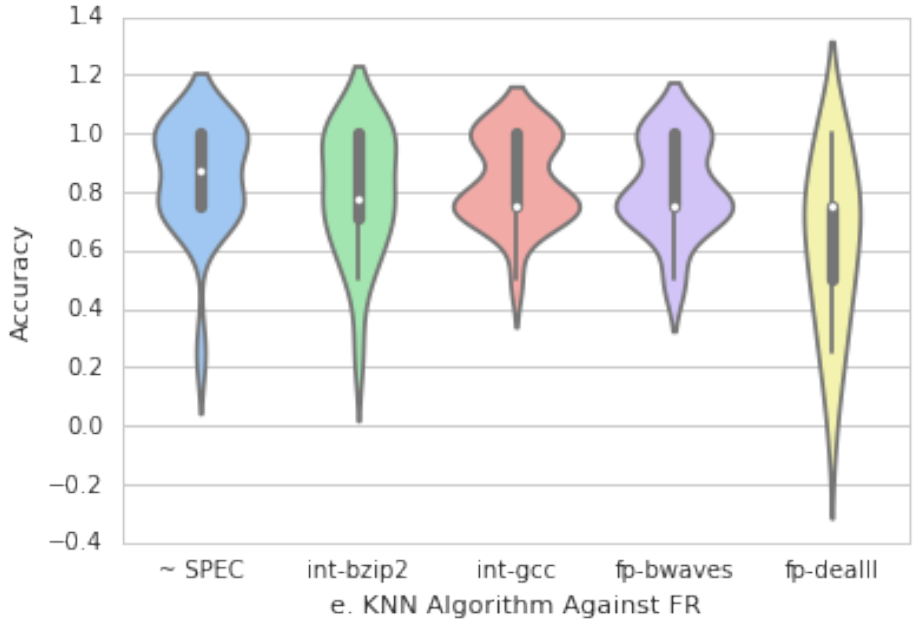


(a) PCANN Algorithm against FR

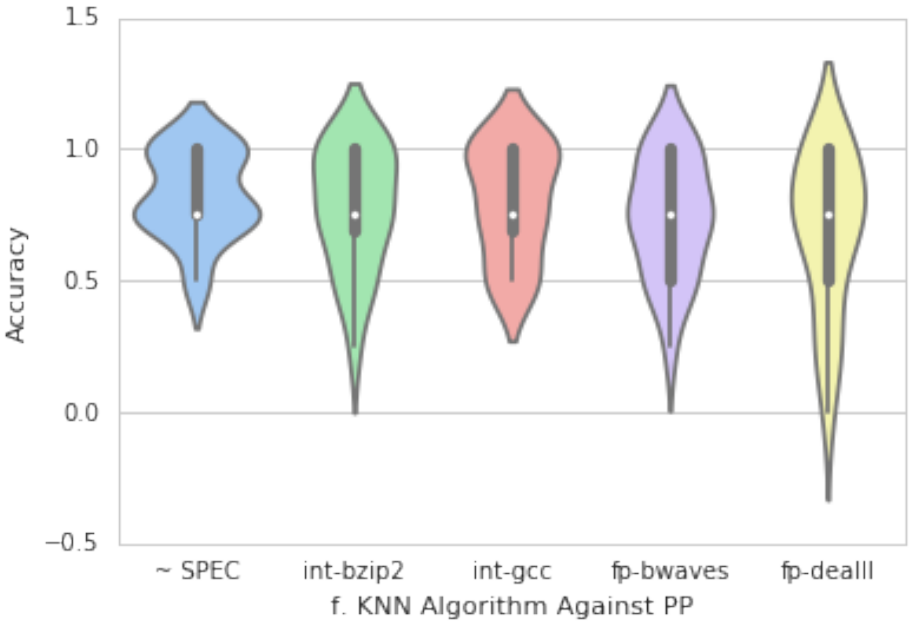


(b) PCANN Algorithm against PP

detecting PP attacks even without benchmarks. k-NN has a similar accuracy rate for FR and PP attacks, but this drops down in a PP attack. The results from C4.5 are therefore seen to be more reliable and robust than from PCANN and k-NN. This is because the C4.5 method deals with noisy data better than the rest of the methods due to fast data exploration, finding the relationships between the most significant variables. Turning to efficiency, the Decision



(a) k-NN Algorithm against FR



(b) k-NN Algorithm against PP

Tree is more inefficient than PCANN and k-NN, but it still detects the attack with reasonable efficiency.

However, running `bzip2` and `gcc` applications in order to load the SPEC benchmark showed that C4.5 has a higher level of accuracy than either NN or k-NN. k-NN's efficiency was slightly lower than C4.5, but NN had the worst accuracy overall. When `bwaves` was

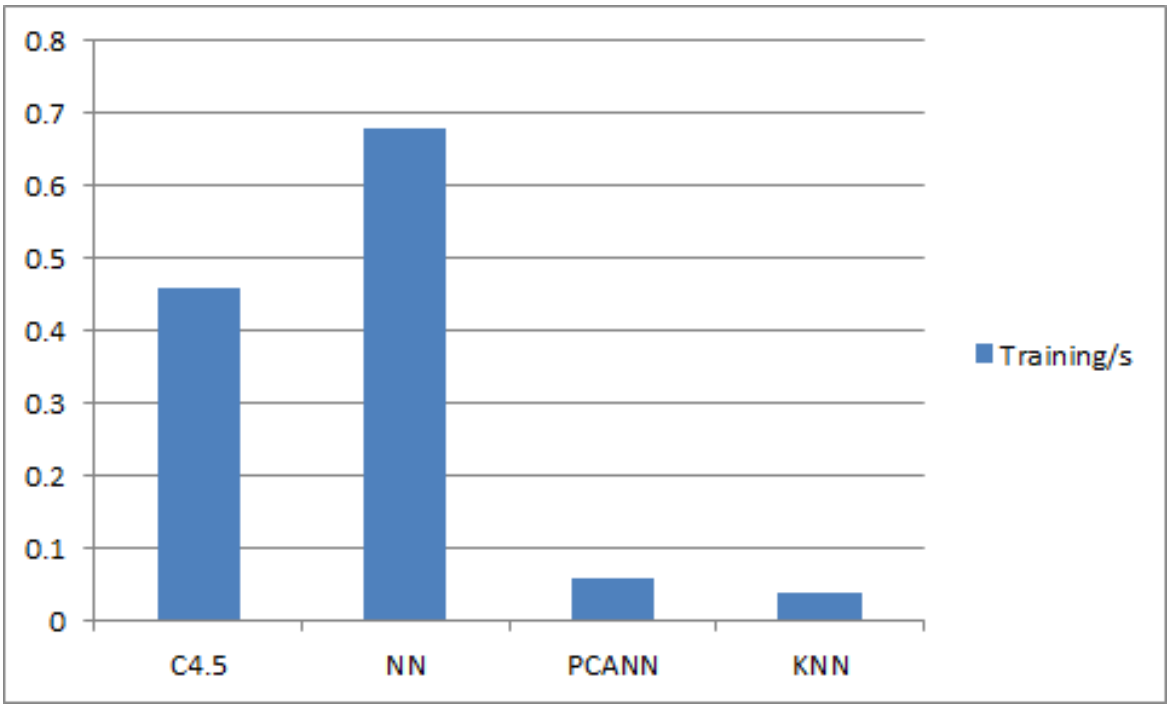


Fig. 3.6 Comparison of time execution for training in selected classifiers

loaded, they all had poor accuracy. This is because `bwaves` is in the float application group and floating operations make heavier use of CPU components than integer operations, resulting in a high number of cache misses and degrading the training of the classification models.

The results shown in Figure 3.6 indicate that the size of the data set will be enough for the detection agent to be able to learn of any malicious activities in a very short time. The worst case is less than 1 second and this compares well with recent and fast FR attacks by 3.6, which needed over 50 seconds to retrieve the entire key. The agent can thus detect the attack early enough to prevent the attacker from stealing the whole key, allowing them to perform the actions necessary to stop the attacker.

It follows that the detection of FR and PP attacks will be difficult in noisy environments, and especially so when intensive floating-point applications are running. This is because floating point applications make use of CPU caches and generate a large number of cache misses. Detection relies partly on CPU cache misses to detect FR and PP attacks.

These methods can be used by a host OS, in both native and cloud systems (though it must be noted that cloud systems are less accurate than native systems), in order to distribute

a fake process running cryptographic algorithms such as AES to identify malicious activities and to prevent them from stealing the entirety of a secret key.

The results show that system activities in the background do not significantly impact the results, with all methods performing well. Intensive workloads introduce more noise into the system, which has a negative impact on accuracy. In particular, the SPECfp benchmark made the result worse than the SPECint benchmarks. This is because the floating operations cause a high occurrence of CPU events. Loading the SPEC benchmarks places stress on the CPU components, particularly on the caches. A SPECfp benchmark interferes with the monitoring processes and introduces noise into the environment.

Chapter 4

Designing and Implementing the Framework (TrapMP)

This chapter introduces a new framework for securing the computational environment from Side Channel Attacks in host Operating Systems (OS). The framework utilises hardware features, namely High-Performance Counters (HPC), which are mainly used to optimise the performance of programs or systems, in both native and cloud systems. The key to this framework is that processor core level observation is deployed to detect abnormal memory contention (or transaction) activities, which are unintentionally performed by the attackers, during real-time program execution. Currently, existing solutions designed to detect side channel attacks perform analyses by relying on the synchronisation of workloads between attack and victim activities. Our approach yields a high-rate detection accuracy with significant performance improvements as demonstrated by the SPEC benchmark in the KVM environment.

The attacker then uses the CPU performance counters to measure differences in memory access, before interpreting the meaning of cache uses by the target's program when both attacker and target are using the same program. In the side channel attack, the attack program runs malicious loops that continuously scan the system's memory to deduce what part of the shared memory the victim is accessing and then builds a statistical model to deduce and retrieve bits from the secret key used by the victim's program. Using HPCs makes it possible to extract features from the attack program's activities, and it is, therefore, essential to establish how such features can be extracted in real-time from program execution attributes.

4.1 Motivation

Anti-virus software and other anti-malware tools struggle to detect side channel attacks, because side channel attacks do not achieve their ends by escalating system privileges. Side channel attackers make use of the current state of the CPU of interest, more specifically memory contentions, known as cache misses. HPCs are the core of CPU components, indicating the current state of a CPU. However, HPC utilisation most often requires kernel or OS privileges either for attack or for defence, but side channel attacks involve abusing CPU cache memories (L1, L2 and LLC) to detect sensitive memory transactions and then access them without OS involvements. As a result, side channel attack programs produce unintentional memory contentions during their execution. Thus, these memory contentions can be monitored with HPC support with minimum requirements, without, for example, system settings and configurations, dedicated hardware resources (Zhang et al., 2016a) or injecting the sensitive applications (Kulah et al., 2018), such as cryptographic applications which are targeted by attackers. Instead, the detection system can be implemented as a service and deployed into the host OS with very low performance overheads.

4.2 Components of Computational Environment

This section describes the necessary components for building the security framework (TrapMP) to detect malicious processes in both native and cloud systems. This section helps to understand how an attack is achieved, what causes the vulnerabilities and in what settings, and from what configurations the attack benefits; it also considers how the program phase is utilised and how it can be used in data collection.

4.2.1 Multi-core Platforms

Mainstream microprocessors support a large number of inter-connected cores with complex memory systems within the CPU dies. Each processor core has private caches L1 and L2 and one inclusive Last Level Cache (LLC) across the processor cores. Any communications between processor cores and other sources in the machine must pass through processor cache memories. Thus, high frequency hardware contentions occur in all cache levels, particularly in LLC, because it represents the highest level of cache memory and is inclusive for L1 and L2. The main source which supplies the CPU with data and instructions is the main memory. Memory access works hierarchically in the sense that each lower level of memory buffers data from one level higher above it. The speed of cache memories varies from level to level.

This make the data leakage feasible at each level. Microprocessor industries have provided flexibility in using CPUs by modifying the hardware settings in multi-core platforms. For instance, an OS can run under process or thread mode. In process mode, two processes cannot share private caches, whereas in thread mode, threads can share private L1 and L2 caches.

4.2.2 Multi-tasking (Model) Systems

Multi-tasking systems have brought together CPU designers and the programming community in terms of their agreement on significantly exploiting the computing power of multi-core processors to save considerable energy with optimal performance. This has motivated researchers to focus more effort on proposing various algorithms and resource adaption (preservation) mechanisms to host the maximum possible number of programs across processor cores with a minimal degradation of system performance. These efforts have led to a heterogeneous computational environment which means that various sort of applications can be accommodated by shared resources such as cache memories. Thus, task scheduling becomes more complex to handle memory transactions.

In real time systems, a program is composed of one or more tasks or processes (for the rest of the thesis we use the term process instead of task), each of which is divided into a number of sub-tasks which are called jobs. Each job consists of a chunk or number of consequent instructions which are queued and managed by the OS scheduler so that they are ready for real-time execution, as each has a time quantum which varies depending on the scheduler algorithms. The scheduler put the jobs in the queue in a fair manner. Each job has an execution time quantum and is bounded by its life-cycle in the environment. The job has to be assigned to one of the processor cores by the OS scheduler. It is not guaranteed that the scheduler will always assign jobs for the same process to the same processor core. Instead, it shuffles them across online (active) processor cores to avoid stalls and this is subject to availability. When a job has been assigned to a processor core, the job has exclusive access to the available resources until the job is complete except in the case that an interrupt is triggered and it has to be suspended. Thus, interrupts have been utilised in the Section Identification Phase 4.13 which traps the attacker processes and changes their execution path into a check point in order to take the necessary action to identify the attackers.

Two or more jobs of the same task can run concurrently across processor cores, but they can never be overlapped or pre-empted (concurrent) in the same processor core. Assigning jobs is on a queue basis. Tasks are given equal time slices called quanta. The jobs of a task

are divided in the time scale in a sequential manner. One job has to be finished, then the next job is assigned to the available cores. When a job is assigned to a core, all the core components are utilised for the job such as HPC, L1 and L2 caches. But the LLC cache is available for all online processor cores, maintaining data until the upcoming jobs evict the previous content. Thus, cross core attacks are viable because it does not matter what accesses the content of the LLC, particularly when two processes are synchronised (i.e. in the case of side channel attacks; go to section 3.3 for more details). This is a limitation of the OS which cannot control memory access at that level, particularly when the resources are shared. As a result, it is crucial to propose a mechanism which supports attack mitigation and is not influenced by any factors such as attackers evading detection systems or hiding themselves from monitoring mechanisms such as Hexpads (Payer, 2016).

Hardware threads are virtual cores which maximise the efficient utilisation of hardware resources. Hyper-threading is a technique which helps to manage the utilisation of shared resources efficiently between threads to achieve optimal performance. This is because, hyperthreading minimises computational latency, particularly in the presence of stalls. When one thread is stalled another thread can be scheduled and uses the resources. The difference between processes and threads is that the cost of loading processes is higher than threads in term of the data structure; more information is held in loading processes than threads. This require more resources and time to be allocated, initialised and loaded. Besides, in hyper-threading, threads need a higher number of inter-communications between them and each interruption requires a context switch. Therefore, high interrupts incur performance overheads to the system due to using system calls and transferring data from memory to registers. Moreover, hyper-threading opens up security gaps in terms of leaking data Zhang et al. (2012) from private cache memory allowing sensitive data to be stolen. This is due to the fact that in multi-threading, the private L1 and L2 cache may be shared. Consequently, the multi-threading feature is disabled by default in many cloud systems such as Amazon AWS, because in this case, processes cannot share private L1 and L2 caches and this prevents side channel attacks from stealing data at this level.

Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space in which L1 and L2 caches can be exploited for data leakage attacks, while for processes running in separate memory spaces, each process has exclusive access to the resources of L1 and L2. LLC and main memory, however, are shared and will be exploited by attackers.

4.2.3 Real-time Scheduling

Scheduling is one of the core OS services to support and mange hardware resources across running programs. The main goal of the scheduler is to minimise power consumption (Zhang and Chang, 2014), which is used by the resources, and offer the optimal performance by minimising stalls (Sherwood et al., 2003a) to provide the optimal dynamic adoption, dynamic voltage and frequency scaling (DVFS)[1] (Valentini et al., 2013). Thus, OS designers and researchers introduce optimal scheduling algorithms to aid bottlenecks and reduce power consumption in order to utilise the highest possible speed that a CPU has taking account of hardware limitations. The main focus in scheduling studies is the efficient usage of underlying hardware resources and how to virtualise and share them across processes. On the other hand, side channel attacks come into account to distort smooth scheduling by misusing the shared resources due to scheduler vulnerabilities while using them (Kocher et al., 2018; Lipp et al., 2018); this results in an obstacle in front of scheduler to scale up CPU components as CPU speed.

In this work, we take advantage of understanding how the scheduler slices the core-based program execution timeline across multiple tasks, and assigns tasks across online cores in the system. So, it is crucial to categorise the scheduler in real-time systems. There are two main types of time slicing that the schedulers rely on in real time systems to manage shared resources: hard and soft real time. For **hard real time**, each periodic task has a deadline for completing its task which means it is restricted to finishing its computations according to timing constraints, and such schedulers are built into embedded systems. In contrast, in **soft real time** there is a deadline for each task, but it is not compulsory to finish the task within the deadline. Instead, it depends on the nature of the program requirement in terms of utilising the resources; some tasks might have a longer life-cycle than the time that has been predicted by the scheduler due to e.g. locality. Consequently, there is flexibility for extending the deadline until the job has been finished. This extension is important for this study, because the profiling is based on the underlying processor core for which it is important to know the behaviour or activities of the jobs and the duration of their assignment in the processor cores. The details are given in Section 4.10.

[1] This is the adjustment process for power and speed settings in various processors in computing devices.

4.3 Program Phase

This section describes the usage of the program phase proprieties in the experiments conducted for the study, in which the HPC is utilised to capture the relevant events according to the *ML*'s execution attributes to visualise the ML activities as phases with the aim of identifying the iterations of the ML's inner loop, which performs the observation of the targeted memory addresses used by victims. This allows the execution analysis models for the proposed framework to be able to capture and predict the potential iterations that might occur per job, and which is assigned to one of the online processor cores, in order to bound and extract the FLUSH+RELOAD attack activities in execution-time.

4.3.1 Program Phase Utilisation

Program phase has been utilised in various computational problems related to performance, such as saving energy (Zhang and Chang, 2014) and performance tuning (Sherwood et al., 2003a). Recent studies have found that the use of program phase leverages CPU scaling, which relies on the correlation between memory and CPU workloads. Program phase provides information to improve scheduler algorithms in the OS. For instance, Skrenes and Williamson (2016) employed the Dynamic Voltage and Frequency Scaling (DVFS) mechanism, which balances high speed CPU with memory access latency to avoid stalls when the workload intends to fetch data from the cache memory. Furthermore, Zhang and Chang (2014) used dynamic configuration CPU frequency to save CPU power. This guides the system to switch the CPU frequency mode into a higher frequency rate when intensive CPU usage is indicated and vice versa. In addition, the program phase has been utilised in performance simulation tools to reduce the simulation time for benchmarks. This is made possible by identifying the sections of code which have similar activities and can be representative of the entire benchmark. Furthermore, the notion of a program phase has been utilised in modern industrial processes such as chemical and biological industries. Such industry systems rely on batch processes to generate products. Consequently, it is recommended that these batch processes be monitored in order to ensure the products' safety, consistency, and reliability. Zhang et al. (2017) used program phase to capture information about faulty cases in the system and feed machine learning algorithms to classify the normal and faulty cases.

4.3.2 Program Phase Definition

In the computational environment, the total computation of any program can be divided into a set of intervals. Each interval is a slice of the program's execution. A set of intervals is composed to form a phase. The phase can occur multiple times within the program's execution. The transactions between two consecutive phases is called phase change Sherwood et al. (2003a).

Program execution behaviours vary from program to program including large-scale ones (Lau et al., 2005). Thus, Dhodapkar and Smith (2003) categorised program phases into stable phase and phase changes. Stable phases are indicated if two or more phases have exactly similar activities, otherwise phase changes are indicated. The presence of phase changes indicates that the phase has been incurred with computational noise. Furthermore, phase detection relies on the nature of a program. The program may be a memory transactions or instruction stream Ding et al. (2006); Sherwood et al. (2003a). In addition, in some circumstances, it is hard to distinguish phase transition; however, selecting relevant events has a positive impact on phase detection by signalling the transactions between phases Ding et al. (2006).

4.3.3 Malicious Loop Phase Modelling

Recall that the main part of a FLUSH+RELOAD attack program body is a malicious loop (*ML*), which steals secret key in AES; inside the *ML*, the two consecutive tasks are executed, flush a targeted memory addresses, which are the range of the memory addresses in which the AES look-up table is stored, using clflush instruction and are followed by access to the flushed address continuously. Thus, in the *ML* structure, each phase has a set of intervals of similar execution activities. The activities are the consumption of the hardware resources such as the L1, L2 and LLC caches. Any `clflush` instruction causes an equal number of cache misses at each hierarchical cache level, when the next access is achieved. As L1 and L2 are private per core, high frequency context switches have more influences on L1 and L2 misses than LLC. This leads to the visualisation of ML phases by noting that LLC cache misses have clear phase transactions between two adjacent (or neighbouring) phases. Figure 4.1 depicts the complete program phase of Flush+Reload from start to end in user space by capturing E1 and E2; and in this case the Flush+Reload program runs for a short period of time for the presentation purpose. However, when utilising Flush+Reload in real systems, the loop boundary is much longer in order to retrieve the entire key bits (as described by Irazoqui et al. (2015)). Figure 4.1 - (a) represents LLC cache misses in a more organised way

than for Figure 4.1 - (b) which is E2. To gain insight into a short period of the Flush+Reload execution in Figure 4.1 - (a), Figure 4.2 magnifies 100 samples out of 4000, in which the phase transactions between every two consequent phases can clearly be seen. Furthermore, choosing a proper sample duration allows profiling to be synchronised with the ML. This feature is useful in the proposed framework because the observation of ML jobs plays a significant role in detection and identification, so it is crucial to make the ML loop iteration activities as distinct as possible from other workload activities.

However, the phases of a program are decomposed into sub-phases in execution-time. Recall that each task of the program is fragmented into jobs, represented (or visualised) in sub-phases, and shuffled with jobs of other programs, so that the jobs are queued by scheduler for execution; and the task scheduler fairly distributes them across online processor cores. Each of such sub-phases appears in between other sub-phases of other workloads in user space. Figure 4.19 shows the sub-phases of the Flush+Reload program which are distributed across other sub-phases of other programs in the same processor core's execution timeline (the execution time is sliced for the mixture of jobs of running programs in user space). The sub-phases of the ML can be recognised across sub-phases of other programs, but the points that indicate the phase transition are hard to capture. HPCs can virtualise sub-phase transactions between two different sub-phases of two independent programs by relying on the ML behaviour based on their execution attributes. This transaction can be used to extract the ML activities across existing workloads in the computational environment. However, defining sub-phases is not an easy task in heterogeneous workload due to changes in program behaviours during run time which are leveraged by dynamic hardware adoption and configuration, but selecting the most relevant and efficient event to characterise ML leads to distinct ML sub-phases across other workloads.

4.4 Threat Model and Assumptions

This section illustrates the potential Flush+Reload attack on microprocessor caches. The attacker exploits hardware and OS vulnerabilities by utilising intentional hardware contentions with a victim's processes, while both the attacker and victim are synchronised, in shared environments in which hardware resources, such as CPU caches, are fairly shared across running applications in both native and cloud systems. The attacker and victim use an AES algorithm to encrypt plain text. An AES algorithm is implemented in `crypto.so`, which is a shared library in an OpenSSL package and it is installed in the host OS Ubuntu 14.04. The attackers can be a malicious program in the host OS or VM in the guest OS. The attacker

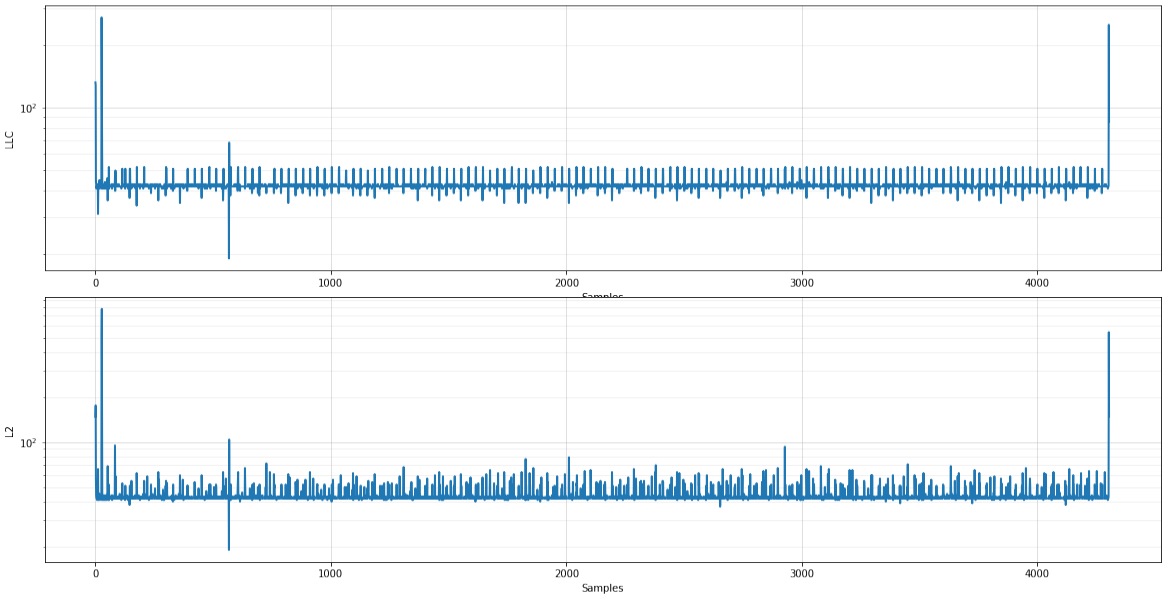


Fig. 4.1 Signature of the attacker program in the native system shows the behaviour of the Flush+Reload program and how it interacts with underlying hardware during its execution

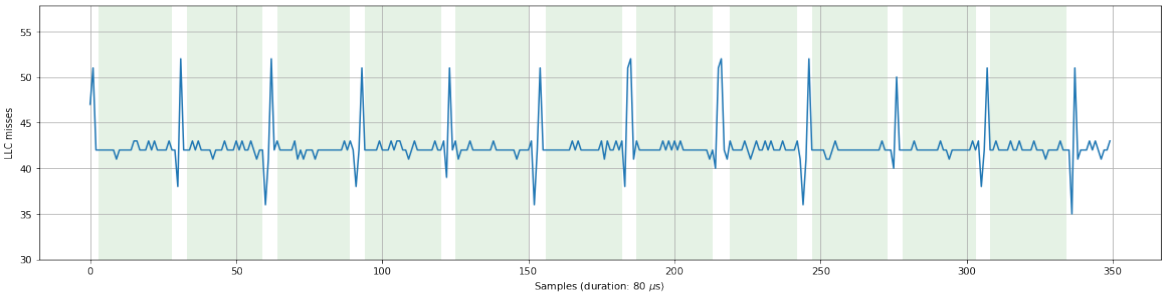


Fig. 4.2 Signature of the attacker program in the native system shows the behaviour of the Flush+Reload program and how it interacts with underlying hardware during its execution and transparently provides interfaces to access HPCs. It is assumed that no malicious bodies have access to the PHCs to modify settings and distort the observations.

analyses the hardware cache contentions to deduce the AES secret keys. More details about the attack thread are given in Section 3.3. Moreover, more than one Flush+Reload attack is running concurrently in the system. Besides, it is assumed that the host OS is trusted.

4.5 The Framework Approach

This section introduces the key components and the main idea of the proposed framework in this thesis, and its influences on the system performance. As has been mentioned in the previous section, side channel attacks make use of a malicious loop to complete the observation task by generating intentional contentions synchronously with the victim on the CPU cache, particularly the LLC. The contentions are the primary source for the attackers to discover information of interest such as the memory transactions of secret elements. On the other hand, most attackers are not aware on their unintentional contentions. Figure 4.1 depicts the attack pattern which is produced from unintentional memory contentions. It is worth paying attention to any ongoing ML in the system. The attack activities in Figure 4.1 are presented in the best-case scenario, in which the only attacker program is running on a specific processor core from the beginning of its execution to the completion. But, in real-time the scheduler time is sliced for all running programs in the system, and they are executed by the CPU in an out of control manner. Thus, it is difficult to distinguish such malicious activities among concurrent programs. For this reason, program phase detection mechanisms have been used to efficiently construct the attack activities, which are sliced and distributed across processor cores, by exploiting the efficient utilisation of the HPCs to investigate the ML activities.

Besides this, the proposed detection system in this study performs detection first then identification. In detection, a supervised machine learning approach is utilised in user-space, whereas in the identification phase, the tasks are deployed in kernel-space. The computational cost in user-space is cheaper than that in kernel-space, because the program executions in kernel-space have higher priority than in user-space, and most of the kernel tasks are interrupt-based. Interrupts incur high performance overheads in the system, because interrupt routines have a higher priority than tasks in user-space in utilising hardware resources. This leads to the system performing a context switch with every interrupt routine call. In a case where the identification has a high number of False Negative cases, a high number of interrupts will be triggered and adversely affect the performance overheads in the system. As a result, efficient classification models are deployed in user-land first to avoid False Negative results. A message will be sent to the identification phase only when attack activities have been

found. In this case, the interrupt routines related to the identifications will be triggered only when the attack(s) are detected. It is therefore essential that the classification model should be sensitive to allow for successful detection of the attack. For this reason, program phase detection mechanisms have been used to efficiently identify the ML loop repetitions and apply a sum of aggregation function to bound the ML's execution attributes in one data point in the data-set before feeding them to the classification algorithms. Consequently, the program phase supports the classifier to be more reliable and robust in detecting the ML iterations.

4.6 Challenges

This section describes the challenges which are addressed in the framework.

Accuracy Accurate side channel attack detection systems must detect side channel attacks with high accuracy. Recent research (Alam et al., 2017; Kulah et al., 2018; Payer, 2016) has suggested machine learning as a good way to detect side channel attacks with high accuracy and very low false positive rates, but the proposed methods rely on synchronisations, in this case, factors like CPU workloads and dynamic hardware configurations (Allaf et al., 2017; Briongos et al., 2017), and smart attacks (Del Pozo et al., 2015) have a negative impact on the accuracy. Notwithstanding, the proposed detection system in this chapter does not rely on the synchronisation approach, instead raw data, which is collected at processor core level, have been used and extract the attack activities by utilising program phase detection mechanisms, which contribute to detection efficiency and reliability, to instruct such activities.

Reliability, furthermore, (Payer, 2016) relies on perf in detection, in monitoring all processes in the system by exploiting a proc file system, which is used by the perf tool. On the other hand, Zhang et al. (2016a) monitor suspected malicious processes only. In both cases, the attackers can escape the monitoring process. In contrast, the proposed detection system does not rely on proc nor on monitoring only suspected processes; instead, every single program execution activity at the processor core level will be captured and inspected to be checked for being malicious or not. At this level of observation, no process activities are able to escape observation, because there is an automatic data collection mechanism.

Identification is a separate process in the detection system. Recently, detection work has focused on detecting side channel attacks without identifying the attacker. Recent works (Gulmezoglu et al., 2017; Payer, 2016) proposed detection techniques but failed to identify which processes or VMs have achieved side channel attacks. However, cloud providers are

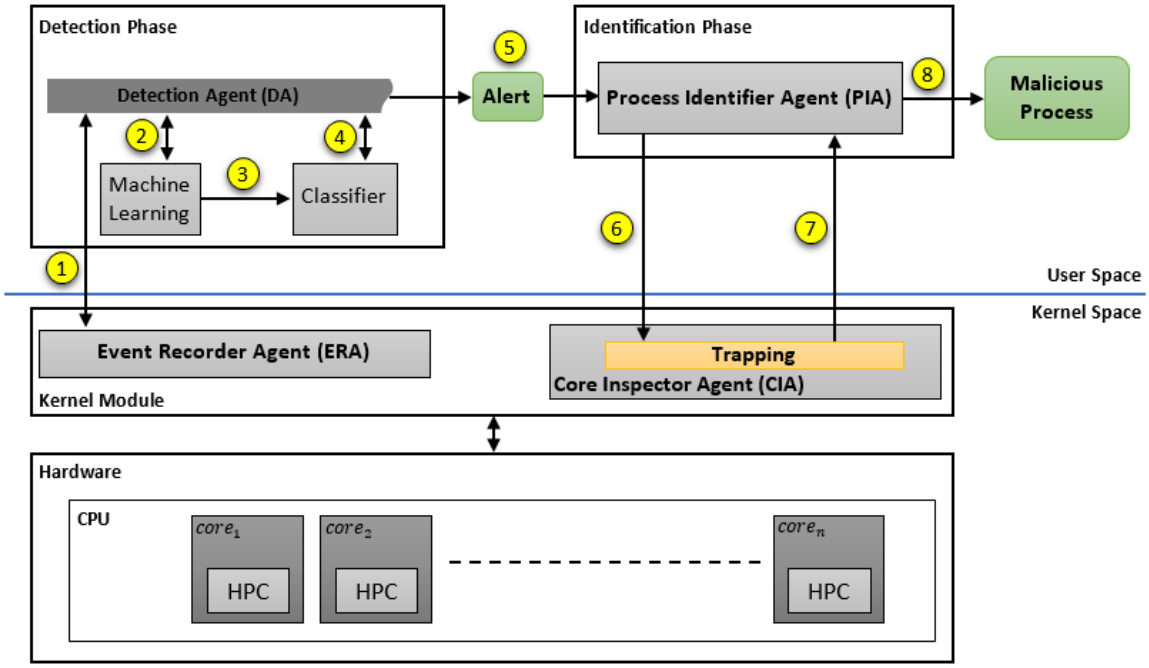


Fig. 4.3 An overview of the proposed framework (TrapMP)

keen to know attackers' identities. The proposed work in this chapter can detect multiple attacks in the system and identify the attacker.

4.7 The Framework Design (TrapMP)

TrapMP is a trapping method for capturing Malicious Processes (MP) at the processor core level. The TrapMP is composed of two parts: the detection and identification phases. Both phases rely on the usage of HPCs to support their models in detection and identification tasks. The detection model is responsible for detecting ML activities in the system, whereas the identification model is responsible for identifying the owner of the ML program. They request information from the kernel module about the state of processor cores, because both the detection and identification models run in user space; and programs in user space have no access to HPCs; the details are given in Section 3.2.3. Figure 4.3 illustrates the high level of the framework, in which the main components are shown, along with their hardware usages and their communications. The yellow notations represent the whole process in chronological order.

1. **Detection Phase:** In this phase, the detection model is responsible for detecting side channel attacks, namely Flush+Reload in the system. The model utilises supervised

machine learning algorithms to classify the attack activities which are achieved by the attacker program in user space. The detection model continuously observes program execution attributes on active processor cores from any ML activities. (1) The Detection Agent (DA) sets up the communication channels with the Event Recorder Agent (ERA) in kernel space to request S samples per processor core. (2) Then DA performs prepossessing of raw data by applying shift and aggregation mean function to combine n of consequent samples to capture ML sub-phases[2]. (3) The DA feeds the new data-sets to the classifier to extract the attacks pattern. (4) The classifier sends back the results to the DA. (5) The DA sends an alert to the identification phase if attack activities have been detected.

2. **Identification Phase:**

   Identification Phase: is responsible to identify the attackers by setting up a trap routine to redirect the malicious program execution path to an interrupt routine. In the cloud settings, the framework is capable of trapping the malicious VMs and identifying them as having the same cost as the native system. (6) The Process Identifier Agent (PIA) requests the Core Inspector Agent (CIA) to inspect any program execution attributes related to the ML execution attributes. This is done by settings and initialising HPC counters to investigate the state of each processor core. (7) If the values of the PMC counters match the attack patterns, which relies on the statistical analysis model, then the CIA will trigger an interrupt to suspend the MP and yield the necessary information about the MP. (8) The CIA reports back details about the identification of the ML to the PIA, Finally, the PIA reports back on the identity of the malicious processes or VMs to the admin users.

4.8 Experiment Setup

Table 4.1 shows the hardware and software specifications for the experiments in this chapter.

4.9 Benchmark

Benchmark suites of programs are given by communities and companies with agreements for them to be representatives and assess the relative performance of a system. They measure performance of a piece of code, an application or a system. The Standard Performance

[2]sub-phase is described in section 4.3

	Type	Specification
Hardware	Machine	HP Proliant DL360G7
	Microprocessor	Intel Xeon X5650 2.66GHz
	Main Memeory (RAM)	16GB
Software	OS	Ubuntu 14.04
	Visualisation Software	Kernel Virtual Machine (KVM)
	Benchmark	SPEC cpu2006 suits benchmark
	Targeted application	AES in OpenSSL implementation

Table 4.1 Hardware and software specifications

Evaluation Corporation (SPEC) CPU2006 Benchmark suite (Henning, 2006) comprises 29 programs each representing a specific program type. For example, bzip2 represents compression programs and GCC represents compiler applications. SPEC CPU2006 is mainly used to evaluate performance for the new generation of computing systems and publish them[3]. However, it has been widely used in program phase problems such as detecting their phases in the computational environment (Sandberg et al., 2013; Sherwood et al., 2003b; Zhang and Chang, 2014) and evaluating energy efficiency (Skrenes and Williamson, 2016). Furthermore, it has been utilised in security domains. For instance, SPEC CPU2006 is used to measure the accuracy of side channel attack detection systems Allaf et al. (2017) and malware detection systems Malone et al. (2011). In this study, the SPEC CPU2006 suite has used for testing the accuracy of detecting and identifying the attacker, and measuring the incurred performance overhead by the proposed framework for the host OS.

4.10 Data Collection

In the experiment settings of this chapter, the data collection relies on HPCs to profile the program execution attributes per processor core based on the shared execution time line among processes in the system. As typical modern microprocessors have seven counters to record the current state of each processor core, seven features of the program execution attributes can be captured concurrently. The CPU of the machine which is used in the experiments has seven PMC to record the processor core state. Thus, only seven events are used to record memory transactions, stall and the number of cycles including three fixed function counters and four programmable counters which are listed in Table 4.2. As the PMC registers are auto counters, the duration is needed to indicate for how long the registers holds the counting. This interval is very critical because the TrapMP wants to

[3]https://www.spec.org/cpu2006/results/

extract individual iterations of ML. The length of time taken to complete each iteration in ML in a Flush+Reload attack is approximately $\approx 0.02\mu$s. A kernel module is implemented to provide an interface between TrapMP and the MPC registers. As TrapMP is composed of two phases – detection and identification phases – each has different settings for collecting data. In the detection phase, the number of samples are much more than in the identification phase ranging from 1000 to 4000 samples for detection and 5-20 samples for identification. The detection phase continuously requires data, but the identification phase needs data when it is informed that an attack has occurred in the system. Furthermore, to collect only program execution activities in user-land, the OS and kernel execution activities will be excluded in the counting by setting the bit number 16^{th} for MSR registers to 1 (for more details on how to set up PMC registers refer to Section 3.2.3.4).

PMCs	Annotation	Events
Programmable	E_1	LLC Misses
	E_2	L2_RQSTS.ALL_CODE_RD
	E_3	L2_RQSTS.DEMAND_DATA_RD_HIT
	E_4	L2_RQSTS.ALL_DEMAND_DATA_RD
Fixed	E_5	Inst_Retired.Any
	E_6	CPU_CLK_UNHALTED.CORE
	E_7	CPU_CLK_UNHALTED.REF_TSC

Table 4.2 Relevant events to side channel attack

The data are collected under different scenarios, for the purpose of labelling classes (normal and attack) of the data-set; the selected programs for the experiments need to be run alone in user-land to recognise their patterns. The target application in the experiments for this chapter is the Flush+Reload attack program. Flush+Reload attack programs have been run and assigned to a specific processor core by using affinity functions `set flush_reload -p 2`.

4.10.1 Data Labelling

The collected data need to be labelled before feeding them to the classification algorithms. In this study, binary classification is employed to classify malicious and benign behaviours. All workloads in user space are considered benign except the side channel attack program activities which are classified as malicious behaviour. To distinguish malicious behaviour, the Flush+Reload program is run multiple times alone in user space to ensure that the profiling records only the Flush+Reload program. Moreover, the `taskset -c` command is used to

Algorithm 1 Data Collection

```
1: procedure DATA_COLLECTION()
2:     setevents(PMCs)
3:     int P,C,S,d
4:     samples[P,C,S]
5:     for each processor p in P do
6:         for each core c in C do
7:             for each s in S do
8:                 reset(PMCs)
9:                 wait(d)
10:                samples[p,c,s] = read(PMCs)
11:            end for
12:        end for
13:    end for
14:    send(samples)
15: end procedure
```

Algorithm 2 Detection Algorithm

```
1: procedure DETECTION()
2:     while True do
3:         recv(samples)
4:         temp = aggregation(samples)
5:         alarm = classifier(temp)
6:         if alarm then
7:             signal()
8:         end if
9:     end while
10: end procedure
```

pin the Flush+Reload program to a specific processor core and request agent to monitor the specified processor core.

4.11 Feature Selection and Thresholds

This section describes how to choose program execution attributes to identify the Flush+Reload attack activities in RTS by utilising HPCs. Further, selecting the most relevant events to *ML* has efficiently affect in detecting and identifying the attack activities.

In this study, the main data collection source is HPCs which is available in modern CPUs. In a typical Intel microprocessor, HPCs support monitoring of hundreds of CPU-related events. These events characterise program execution behaviours. However, these events are not equally beneficial to address a specific problem. For instance, some of the events might visualise the Flush+Reload execution attribute, such as L1, L2 and LLC misses, which are more sensible than other events as shown in Figure 4.1, LLC clearly present the phase transaction between every two consecutive iteration in ML. They therefore need to be examined to find the most efficient events which offer better solutions to the problems. Recent research Alam et al. (2017); Briongos et al. (2017); Zhang et al. (2016a) have used machine learning algorithms to choose the most efficient events to support detection models, but in this study, the events are chosen relying on the nature of the attack programs, particularly ML in which the L1, L2 and LLC cache misses are equally be triggered, and the window size, which indicate the interval that the specified events need to be counted. Therefore, in the following subsections, Descriptive Statistic Functions (DS) are used to analyse the run-time execution behaviour of the attack program to determine the thresholds and relationship between the selected events which describe of the program execution attributes.

4.11.1 L1, L2 and LLC Misses Are the Best Features to describe *ML* activities by Flush+Reload Attack programs

Feature selection in detecting side channel attacks at processor core level is critical because in RTS it is hard to predict the possible workloads; and get the same observation of a program's execution activities due to randomness of resource assignments across processes by OS scheduler. Consequently, it is crucial to find out events which virtualises the attack precisely and less affected by those problems.

In this study we focus on the ML inside Flush+Reload program, which is the core part of the program that efficiently explore the vulnerabilities. Thus, the main task of each iteration

in ML is composed of `clflush` instruction and followed by `mov` instruction. The `clflush` instruction removes the data from the hierarchical caches (L1, L2 and LLC) at a specific memory address, whereas the `mov` instruction then accesses the flushed memory address from main memory. The access to the flushed memory address requires n misses for each cache level. So, we assume the n misses of (L1, L2 and LLC) will be occurred while the jobs of ML is assigned to one of the active processor core. Consequently, a very strong co-relation among L1, L2 and LLC caches can be noticed. This is the key intuition in the proposed framework to detect and identify the attacker at processor core level observations.

However, `clflush` might be used by the operating system's Memory Management Unit (MMU) when MMU is not sure what to do with dirty cache lines, OS uses `clflush`. This makes it possible to incur noise in the observations. But, Section 4.10 addresses this problem and discusses ways of excluding the OS activities. Furthermore, noise may also interfere with observations if `clflush` is used in user space by another program, in this case, causing `clflush` instructions that were not initiated by the attack program to be visible in the observations. It is, however, possible to identify the malicious loop with great accuracy because of one of its particular characteristics: its repetition of not less than 25000 iterations, which being the minimum number of operations required to retrieve every bit making up the whole key in native systems (Irazoqui et al., 2015).

4.11.2 Descriptive Statistics to Describe Program Executions

Descriptive statistics are used to summarise experimentally generated data-sets for use in feature selection and threshold identifications. Consequently, this section introduces descriptive statistics as a mathematical tool to compare the execution attributes of the attack program and SPEC workloads to show their low-level activities in fine-grained details and to show how the DS supports the detection and identification models by identifying the thresholds and relationship between features in the framework.

4.11.2.1 Descriptive Statistics

The main uses of descriptive statistics are to describe data-sets' central tendency, variability and distribution. A data-set's tendency is found using mean and median, while the distribution's variability and degree of skew are measured using min, max, variance and standard deviation. A comparison of the program execution attributes' statistical outputs allows us to determine the program's degree of uniqueness during program executions in real-time systems. Each statistical property has both strengths and weaknesses. As an example, the outlier

is the commonest descriptive statistics problem. It has negative impact on measurements, especially median and variance properties. The use to which descriptive statistics and their errors are put depends on the nature of the case study. This study shows not all descriptive statistics properties to be equally useful in the analysis of program execution attributes, and so different properties are used by different features. Mean and standard deviation are mainly used and they can be represented mathematically as follows:

$$\delta_F = \sqrt{\frac{\sum_{f=1}^{F}\sum_{n=1}^{N}(X_{f,n} - \bar{X})^2}{N}} \tag{4.1}$$

When:
δ_F = Standard deviation of feature f
$X_{f,n}$ = i^{th} sample in f^{th} feature
$\bar{X}$ = the mean of $X_{f,i}$
F = number of features
N = number of samples
To get mean of each feature $\bar{X}$

$$\bar{X} = \frac{1}{N}\sum_{f=1}^{F}\sum_{i=1}^{N}X_{f,i} \tag{4.2}$$

The features are analysed individually to extract program execution attributes in both native and cloud systems, and this is where descriptive statistics are useful. The next section analyses the program execution attributes using descriptive statistics tools.

4.11.3 Defining Thresholds

This section illustrates selection of candidate features and thresholds as parameters in the detection and identification model to contain the attacker program, and fleshes out the illustration using Descriptive Statistics to analyse program execution attributes.

4.11.3.1 Distribution

Section 4.4 showed that (L1 and LLC) cache misses are the optimal events describing Flush+Reload activities, particularly ML. Ten experiments were conducted each with 1,000 samples and only the attack program was running, and it was pinned to a specific processor core. The distribution of L2 and LLC cache misses of the attack in native system is shown in Figure 4.5, where the area beneath the red line shows the LLC misses distribution and the

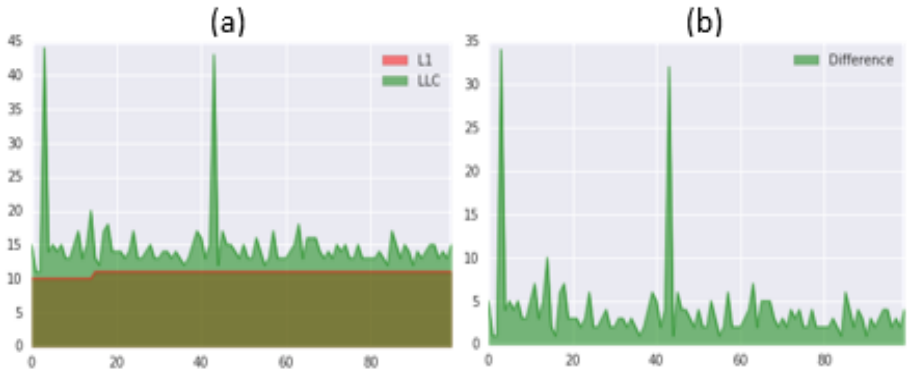


Fig. 4.4 Is the different L3 and L1 cache misses which is considered as noise

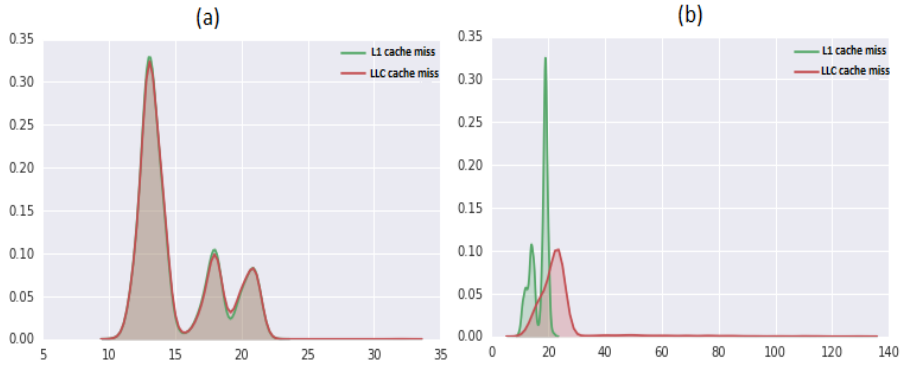


Fig. 4.5 L1 and L3 cache misses distribution of the attacker's program in cloud systems

area under the green line is L1 cache misses distribution. They are almost congruent with each other, showing that they are issuing almost the same cache miss rates.

As shown in Figure 4.5, LLC is still constant when the attack program runs with SPEC workloads, but there is a slight change in L2 cache misses. The workload variation is shown in Figure 4.5 (a), where the area inside the red line is the original cache misses, and the area inside the green line represents noise filtered out by E_2 event.

However, turning to cloud systems, Figure 4.5 (b) shows poor overlap between L1 and LLC, and this is the result of noise causing misses in L1 and L2 cache because of the VM's extra translation layer. As already discussed, the *L2_RQSTS.DEMAND_DATA_RD_HIT* event can remove the noise. In Figure 4.5(b), the area under the green line shows the noise from L1 cache misses penalty and the area under the red line is the actual L2 and LLC cache misses penalty.

4.11.3.2 Program Execution Instability - Tendency

Due to the runtime dynamic optimisation system, the OS utilises $clflush$ instruction to remove stale translation from the cache Bala et al. (2011). The purpose of the tendency is to address the variation of the program behaviour in real time systems.

Understanding program behaviour is at the foundation of computer architecture and program optimization. Many programs have wildly different behaviour on even the very largest of scales (over the complete execution of the program). During one part of execution a program may be completely memory bound while in another it may always be stalling on branch mispredictions. Due to this time-varying behaviour of programs.

In profiling, the most obvious obstacle is the instability of run-time program executions, which varies between experiments even though the configuration is the same. Sherwood et al. Sherwood et al. (2003b) investigated that the program behaviour can change many time. For instance, some programs in real systems are stable , but some are not such as in SPEC CPU2006 bizp2 has more stable program phases than gcc (Sherwood et al., 2003a). This can mean missing an attack program's real-time activities when attempting to extract the execution attributes of the attack program from the observed data. Tendency is the way to overcome this problem. Twenty experiments were conducted to show possible LLC cache miss patterns. LLC was chosen for reasons set out in section 4.11.3.1: LLC does not change with changing workloads. A variation was seen from run to run, and the problem was addressed using tendency. Failure to handle unsuitability properly can reduce the likelihood of detecting malicious processes, because patterns that take time to present themselves need the matching algorithm to wait until the pattern is seen. Failure to handle unsuitability properly can reduce the likelihood of detecting malicious processes, because patterns that take time to present themselves need the matching algorithm to wait until the pattern is seen.

Figure 4.6 shows a range of frequencies for LLC cache misses from 9 to 22 in this setting. These figures were collected from twenty experiments sorted into three groups ($G1$, $G2$ and $G3$) as shown in Figure 4.6. Each pattern ($G1$, $G2$ or $G3$) occurs whenever the experiment is run. Each group has a range of figures indicating that they can be obtained from the counters during the experiment. In this experiment, $G1$ occurs more often than $G2$ and $G3$ – but that should not be taken as indicating that this always happens. It can also be seen that each group comprises three or four figures, one of which is a candidate for the highest frequency. Group $G1$ has four figures in the range 11, 12, 13 and 14. The highest in the group is 13, meaning that 13 is more likely than the others to be the one that occurs when the pattern is captured.

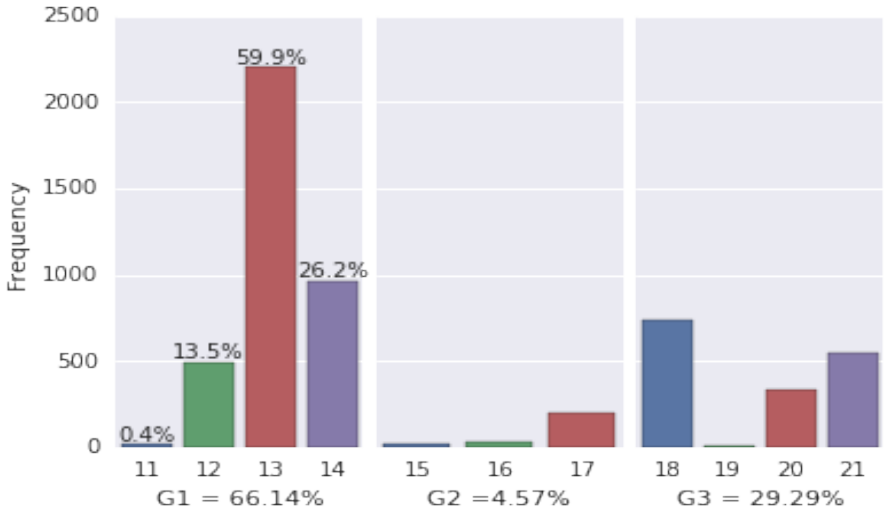


Fig. 4.6 L1 and L3 cache misses tendency of the attacker's program

4.11.3.3 Comparison of Feature Variability

In the identification phase, the matching algorithm looks for a feature for which the variation is as close to zero as possible to form the starting point for checking for an attacker program's presence on a processor core. Variability is measured using Standard Deviation (STD) in order to find the events in the attack program with the least deviation across different workloads.

Native Systems: In Table 4.3, the STD column shows the STD of four programmable counters, counting (L1 Instruction hit, L1 cache misses, L2 cache misses and LLC cache misses). As will be seen, the lowest STD (≈ 0.25) belongs to L3 cache misses of the attack program running alone in a native system. Row FR+SPEC shows the events under four SPEC applications. Except for a slight increase in L2 cache misses, the data shows no significant variances.

Cloud Systems: In Table 4.4, column STD gives the STD of four programmable counters: L1 Instruction hit, L1 cache misses, L2 cache misses and LLC cache misses. The lowest STD (≈ 0.25) was for LLC cache misses of the attack program running in a cloud system. When the attack program runs in cloud systems together with SPEC benchmarks, LLC has the lowest STD at 2.

In the Cloud settings, the lowest STD is for `CPU_CLK_UNHALTED.REF_TSC` and the highest is for *INST_RETIRED.ANY*. The STD of fixed counters in both native and Cloud settings are might higher than for programmable counters, and so the identification phase cannot rely for detection on fixed counters.

Programs	Events	Statistics			
		Mean	STD	Min	Max
FR only	E_1	0.036524	0.250817	0	11
	E_2	15.166787	3.018877	11	32
	E_3	0.008996	0.631708	0	47
	E_4	15.130263	3.0017	11	22
FR+SPEC	E_1	0.036524	0.250817	0	11
	E_2	15.166787	3.018877	11	32
	E_3	0.008996	0.631708	0	47
	E_4	15.130263	3.0017	11	22

Table 4.3 Describes the necessary statistics to support statistical analysis to find the best features and thresholds. This statistic is the key for process identification. The data is collected in a native system and outliers are removed

Programs	Events	Statistics			
		Mean	STD	Min	Max
Cloud	E_1	1	2	3	4
	E_2	23.61	10.93	11	130
	E_3	3.88	14.79	0	252
	E_4	16.92	2.90	10	22

Table 4.4 Describes the necessary statistics to support statistical analysis to find the best features and thresholds. This statistic is the key for process identification. The data is collected in cloud system and outliers have been removed from the data

In summary, this analysis shows LLC and L2 misses to be the two candidates that can be used in the matching algorithm, because their values are identical except when SPEC applications also run, in which case L2 changes with a very low variance. Using L2 hit allows this variance to be filtered out from L2 misses.

4.11.3.4 Min and Max

The thresholds of fixed counters are found using Min and Max properties. Although profiling results suggest that this range can be inferred from other workloads, the logical approach is to use the fixed counter thresholds after four programmable counters have been checked. Figures 4.7 shows the ranges of three fixed counters using Min and Max priorities in native and cloud system respectively. Changing the workload changes the range.

Programs	Events	Statistics			
		Mean	STD	Min	Max
Native	E_5	20872	5751.47	1200	36939
	E_6	18608	4607.69	2107	29894
	E_7	18594	4613.49	2124	29878
Cloud	E_5	29907	5751.47	1500	36939
	E_6	26567	4607.69	2107	49894
	E_7	26586	4613.49	2124	49878

Table 4.5 Describes the necessary statistics to support statistical analysis to find the best features and thresholds. This statistic is the key for process identification. The data is collected from native and cloud systems and outliers have been removed from the data

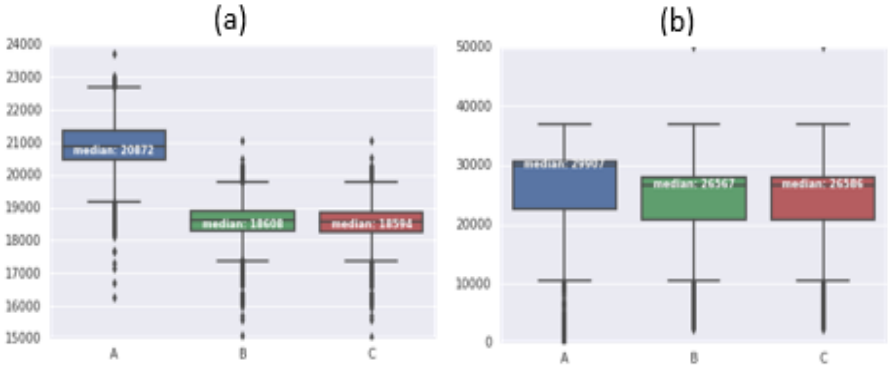


Fig. 4.7 Min and Max of each fixed counters of attacker program in native system

4.12 Detection Phase

This section describes the detection phase of the framework. In this section, the detection models are described and the machine learning algorithms are utilised with the comparison to three classification techniques.

4.12.1 Moving Window Aggregation (MWA)

The aggregation mean function is employed to find the related samples which belong to a single job in ML. As a scheduler cannot be controlled in terms of assigning jobs to the online processor cores and the duration of the assignments, the default is to guess how many samples belong to a job and for how long a processor core holds the job. Consequently, giving raw data to the machine learning algorithms will negatively impact on the performance of the classifiers in detecting side channel attacks. Thus, we leveraged the MWA algorithm.

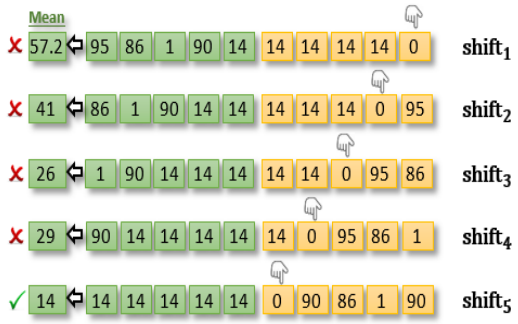


Fig. 4.8 Aggregation and five shifts of the data-set. $shift_5$ represents the best aggregation which captures the whole samples of one of the *ML* jobs, which is counted as an attack activity

To construct the phases of the ML by finding the set of consequent samples in execution time, which belong to the ML jobs.

MWA is the process of partitioning the data-set D which has N samples with F features into subsets $\bar{D}$. Each subset contains the consequent n samples, when $n \subseteq N$, and F features; and averages them to produce one sample which represents one *ML* phase. As a result, a new data-set $\bar{D}_i$ will be generated with the length of $\frac{n}{N}$ samples. This is to transform each the ML phases n into one sample and this will be classified as attack activities. Still it is not guaranteed that the whole body of the phase will be captured, because there might be a sample from the neighbour jobs of other workloads which will interfere with the ML phases. To overcome this problem, the original data-set will be shifted n times and the same procedure split and the mean function for each subset will be repeated to generate n of $\bar{D} = \{\bar{D}_1, \bar{D}_2, .., \bar{D}_n\}$, where n is also the threshold which indicates less than the maximum length of the ML samples which might appear in each ML phase in real-time. The whole data-set will be given to the classifier to allow more chance to detect any potential ML activities. The MWA algorithm provides reliability and robustness in the detection system, because it tries to extract and capture each ML phase by combining the chronological sequence of samples which belong to each ML job in execution-time.

Figure 4.8 illustrates 10 samples of the original data-set with F_i representing LLC cache misses when $F = \{F_1, F_2, F_3, ..., F_f\}$ $f_i \in F$. The same operation will be applied to each F_{i+1} *when* $i = \{1, 2, ..., f\}$. This figure consists of five sub-figures each representing one round of the shift to generate n new $\bar{D}_i$. The single column on the left-hand side represents the mean function of n samples, when $n = 5$ in this example. With these execution settings, the LLC cache miss is equal to 14; more details are given in the previous section on how to choose the LLC cache miss. Thus, the average of n samples should be 14. In $shift_5$ the five

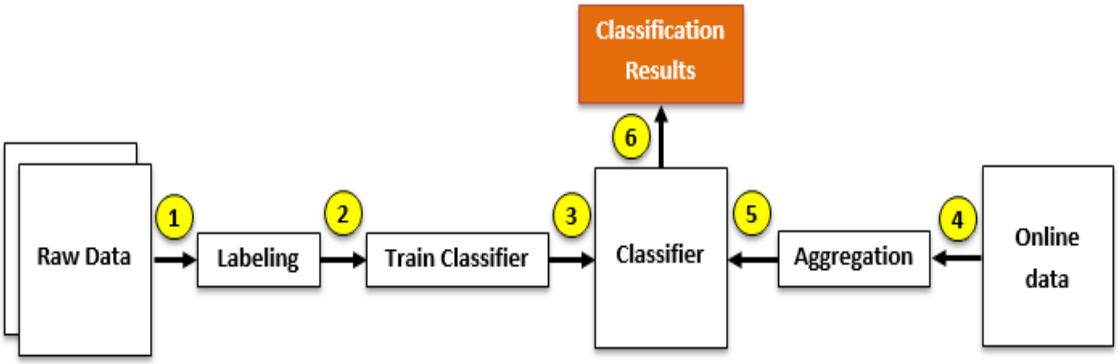


Fig. 4.9 Detection model overview

green samples are aggregated and averaged to 14. Thus, 14 represents one of the ML phases in real-time.

4.12.2 Detection Model Overview

Figure 4.9 illustrates the detection systems, of which the general concept is described in this section. Step ① : the detection system communicates with the Event Recorder Agent (ERA) for the collection of data from the PMC counters. The ERA is a model-based implementation because the PMC counters can only be written to with the kernel's permission. The ERA profiles all the jobs which are currently assigned to the online processor cores as raw data. Within a loop inside ERA, the samples with a specific interval will be recorded. The duration of the loop is related to a possibility that the attacker cannot escape observation. For instance, if the attacker requires approximately one minute, as is the case in (Irazoqui et al., 2014), the ERA starts profiling at half the time of the required time which the attackers need to retrieve the whole secret key and feeds this to the DA for classification. Step ② takes place inside the DA; the off-line data need to be prepared by applying the MWA 4.12.1 algorithm aggregated with average function and then labelled relying on the thresholds, which are indicated in Section 4.11.3. After labelling, the data-set will be split into train and test data-sets. Step ③ the training data-set is then fed to the machine learning algorithms including (k-NN, C4.5 or random forest) to build the classifier ④. Step ⑤ the online data will be collected again. Step ⑥ then the new data are prepared by applying the MWA algorithm, but without labelling them. Step ⑦ the classifiers predict the unseen samples to detect any attack activities in the system.

Algorithm 3 Detection Algorithm

```
1: procedure DETECTION()
2:     while True do
3:         recv(samples)
4:         temp = aggregation(samples)
5:         alarm = classifier(temp)
6:         if alarm then
7:             send_signal()
8:         end if
9:     end while
10: end procedure
```

4.12.3 Methodology

In the previous section, synchronisation-based detection of side channel attacks was conducted by comparing three machine learning algorithms PCANN, k-NN and Decision Tree algorithms in which the decision tree outperformed the k-NN and PCANN under various workloads, light and heavy, to notice how the workloads influence the classifier's performance in terms of detecting side channel attacks. In this chapter, the side channel attack program will be monitored at the processor core level without the detection being synchronised with attacker programs. In this approach, three different supervised machine learning algorithms have been used by considering the single tree algorithm C4.5, see Section 3.4.4, light weight algorithm k-NN, see Section 3.4.3, and random forest, which will be described in detail in conjunction with the limitation of single tree algorithms in this section.

In the previous chapter, we investigated how Decision Tree algorithms outperformed other algorithms. However, Decision Tree (DT) algorithms have limitations in terms of accuracy and efficiency like any learning algorithms. The common problems in Decision Tree algorithms are well known as relating to over-fitting and under-fitting, which are described in Section 3.4.0.2. With overfitting, the classifier is trained very well on the training data-set, but cannot be generalised to the testing data-set. This means that Decision Tree algorithms are sensitive to the specific data-set on which they are trained on, but fail for a testing data-set. This is because the algorithm takes into account every data point including outliers in the training data-set as shown in Figure 4.10. Consequently, the classifier is only poorly generalised to the new samples. In contrast, under-fitting is missing important structural information about the sample space due to insufficient training samples. Therefore, Breiman (1996b) decomposed the over-fitting error into bias and variance to overcome such problems. Single tree-based algorithms have low bias and high variance. High variance

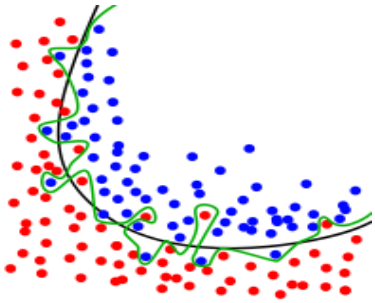


Fig. 4.10 Over-fitting problem on training data-set

affects detection accuracy negatively. However, constraints (e.g. length of the tree) can be used to optimise the tree model accuracy, but single tree predictors suffer from generalisation problems.

As a result, Breiman (1996a) introduced ensemble methods to avoid the variance problem by constructing many prediction models and training them with subsets, which are produced by splitting data points in the original data-set, and combining their decisions to improve accuracy and robustness over the prediction of an individual model. Ensemble methods are mainly categorised into bagging, which is based on randomisation, and boosting techniques (Geurts and Louppe, 2011). In spite of the fact that the boosting technique has gained the attention of researchers and has been applied in various domains successfully (Sayadi et al., 2018), the focus in this study is on the bagging technique.

Bagging (short for Bootstrap aggregating) was introduced by Breiman (1996a) is an ensemble learning technique to decrease the variance of a predictor by bootstrapping samples with replacements from the original data-set to train prediction models of any supervised machine learning algorithms and aggregating their results to select the best predictor. Algorithm 4 explains the steps of a typical bagging algorithm with minimum requirements.

The bagging algorithm, firstly, enrols a constructive loop to generate NL subsets in a random space (choosing randomly from replacements) from the original data-set D and learner algorithm LA of the same algorithm. Secondly, after generating a vector of LA, the aggregation function selects the best learner C_{best} from the LA. Since Bagging is utilised in classification and regression problems, the aggregation Max function is used for classification problems to select the best learner who has the most votes among internal predictors; thus, aggregation is calculated in Equation 4.3, whereas, the aggregation Mean function is used for regression problems to average the results of the learners L; thus, aggregation is calculated in Equation 4.4. The focus in this chapter is on the classification problem.

Algorithm 4 Bagging Algorithm

1: **procedure** BAGGING_CLASSIFIER(LA,D,NL)
2: LA Learner Algorithm
3: D Original Data-set
4: NL Number of learners
5: C_{best} Final ensemble learners
6: **for** i=1,2,...,NL **do**
7: $S_i = BootstrappingSample(D)$
8: $L_i = LA(S_i)$
9: **end for**
10: $C_{best}(x) = MaxAgr_y \sum_{i=1}^{NL} L_i(x,y)$
11: **end procedure**

$$C_{best}(x) = MaxAgr_y \sum_{i=1}^{NL} L_i(x,y) \tag{4.3}$$

$$C_{best}(x) = \frac{1}{NL} \sum_{i=1}^{NL} L_i(x,y) \tag{4.4}$$

Random forest is the implementation of the ensemble concept by using a bagging technique to construct a collection of Decision Trees. Random forest reduces over-fitting, which is generated by single tree algorithms, by utilising bagging (Barandiaran, 1998) technique. The Bagging technique in the random forest algorithm, uses a subspace randomisation scheme to re-sample, with replacements, the training subsets which are used to grow new individual trees. In the random forest algorithm, the base models are tree structured models. The tree models in the forest can be generated by any Tree algorithms such as CART, C4.5 or ID3.

Breiman (2001), for the first time, introduced random forest to decrease variance, which is generated by a single tree-based predictor, by constructing many internal tree predictors on CART algorithm in the forest, each of which is trained on an independent random sample derived from the original data-set with replacements. Furthermore, each random sample is composed of random features to increase the chance of contributing the maximum number of features in the splitting processes, which is also called diversity. Algorithm 5 illustrates the process of the random forest algorithm for classification problems.

Algorithm 5 Random Forest Algorithm

procedure BAGGING_CLASSIFIER(TLA,D,NL)
    TLA Tree Learner Algorithm
    D Original Data-set

    NL Number of learners
    TC_{best} Final tree classifier
    for i=1,2,...,NL **do**
        $S_i = BootstrappingSample(D)$
        $TC_i = TL(S_i)$
    end for
    $TC_{best}(x) = MaxAgr_y \sum_{i=1}^{NL} (TC_i(x,y)$
end procedure

Firstly, the algorithm enrols a constructive loop to generate a vector of bootstrap samples S from the original data-set D. Each bootstrap sample is denoted as S_i. The tree algorithm (CART) is used to grow a vector of tree classifiers TC on the S_i bootstrap sample. Each S_i is composed of a subset of features which are randomly chosen from the data-set $f \subseteq F$, where F is the full set of the features in the original data-set. Then the new tree nodes split on the best features in f rather than F. Secondly, after generating a vector of TC, the aggregation function selects the best tree classifier TC_{best} from the TC. Aggregation Max function is used to select the best tree classifier which has the most votes among internal predictors.

Class imbalance in binary classification occurs when the majority of class instances (normal activities) outnumbers the minority class instances (attack activities). Previous research has shown that the negative impact of imbalanced problem is present in real-world classification problems (Galar et al., 2012; Li, 2007; Nikulin et al., 2009; Rokach, 2016), particularly in malware studies (Zhang et al., 2016b). Furthermore, for the collected data in the experiments for this chapter, the attacker's ML activities have a lower number of instances than normal workloads activities, which is of interest from the point of view of the learning task. This will lead the classification model to have a negative impact on the classification accuracy of unseen data.

Standard machine learning techniques may be leveraged by the majority class and ignored by the minority class in prediction (Rokach, 2016). This means that the minority classes are ignored to contribute to the classification task. To overcome this problem, Cieslak and Chawla (2008) suggested the use of a base induction single tree algorithm by splitting the criteria in tree algorithms. Furthermore, creating synthetic data-sets is another solution for the

classification problems in imbalanced data-sets. Under-sampling in data-sets reduces the size of the majority of data classes (Japkowicz, 2000), whereas in over-sampling of data-sets, the minority class(es) instances are increased Chawla et al. (2002). On the other hand, Li (2007) used the Bagging technique to resolve an imbalanced data-set without creating synthetic data or making changes to the existing classification systems.

Random forest has many applications for balanced and imbalanced data-sets. It has received a lot of attention from researchers in imbalanced data-sets because random forest encourages diversity (Nikulin et al., 2009). This was achieved by introducing an ensemble approach to the base algorithm by replacing a new splitting criterion and producing many tree classifiers instead of one tree to make a decision (e.g. ensemble approach). A decision as a whole can be used to mitigate the classification in an imbalanced data-set. This approach works by creating a forest of tree classifiers where each individual classifier is trained using a balanced subset, where all minority classes are included with randomly chosen majority classes. The reason for using ensemble methods is to imbalance data-sets (Rokach, 2016).

4.12.4 Experimental Design

The design of the experiments in this chapter can be summarised in two phases. In the first phase, the data collection is performed in native and cloud systems with ten-fold cross validation (CV) executed for each. In each CV, a new data-set is constructed from different data points from the original data-sets then the new data-set is divided into 70% training and 30% testing data so that all data points contribute to the model building stage.

In the second phase, three different machine learning algorithm techniques have been used including a single tree algorithm C.45, a light weight algorithm k-NN and a random forest bagging algorithm. With each CV iteration, the new training data-set is fed to each algorithm to build a classifier and then the new testing data-set is used to evaluate the classifier.

4.12.5 Experimental Results and Analysis

In this section, the results of the experiments are shown for each Decision Tree C4.5, k-NN and random forest algorithm and they are visualised by utilising ROC Area Under Curve (AUC). The figures in this section depict the performance of classifiers in discriminating between two process activities which are normal and attack.

In ROC-AUC figures, the classifiers' outputs are represented as ROC curves. Each ROC represents recall (sensitivity) against specificity at incremental thresholds between zero and one across 10 folds when the same data-set is randomly shuffled, resulting in each fold

having a different spread of data. The Y axis plots the classifier outputs' True Positives Rates (recall) and the X axis plots False Positive Rates (specificity). Each fold is an individual ROC and is represented by a light blue line which is detection quality. The solid blue line is the mean of 10 classifiers. The ideal representation is when the ROC curves have x=0 and y=1. This indicates that the classifiers classify normal and attack classes in unseen samples 100% of the time.

When the classifier has predicted unseen data-sets, its accuracy is evaluated to test how efficient it is at extracting Flush+Reload activities. The two characteristics, recall and specificity, are plotted along a ROC curve. The ROC was put forward by Bradley (1997); Fawcett (2006) to enable performance metrics via a predictive model by drawing line graphs connecting recall and specificity. A point on the curve will signify a ratio between 0 and 1. Since the halfway point on this curve represents a random guess, the diagonal connects the points (0.5,0.5). Anything above that diagonal will be more accurate than a random guess, and the actual position enables its accuracy to be characterised on a continuum from good to excellent, with the very best performance closest to the top right corner. Anything below the diagonal is likely to be even less accurate than a random guess.

4.12.5.1 k-NN Results

Figure 4.11 and 4.12 show the ROC metric that evaluates the k-NN classifiers' ability to detect the ML activities among normal workloads in the host system in both native and cloud settings respectively. Success in observing program execution attributes and classifying processes as malicious or benign, as a measure of the risk of existing side channel attack in the system, is shown as being estimated by the AUC of ROC. The model identifies the ML in a native system with very high accuracy (AUC=0.99 for an average of 10 folds, with a zero-confidence interval) Figure 4.11. In the cloud, however, the same algorithm is trained on a data-set that captured the fact that VM activities were less accurate at predicting malicious activities from among other workloads (AUC=0.96, confidence interval=0.02) Figure 4.12. The classifier is therefore 3% less efficient at identifying malicious loop activities in the cloud than in a native system.

Recall, Flush+Reload frequently repeats the same task, which is organised by executing the clflush instruction to a specific memory address of interest and then executing mov to the same address. When it receives the memory address from which to read the contents, mov must retrieve them from memory pages because the previous clflush rendered the contents in hierarchical cache memory at that address, and invalid contents are updated from main memory leading to a sequence of hardware events. Cache misses at L1, L2 and LLC are the

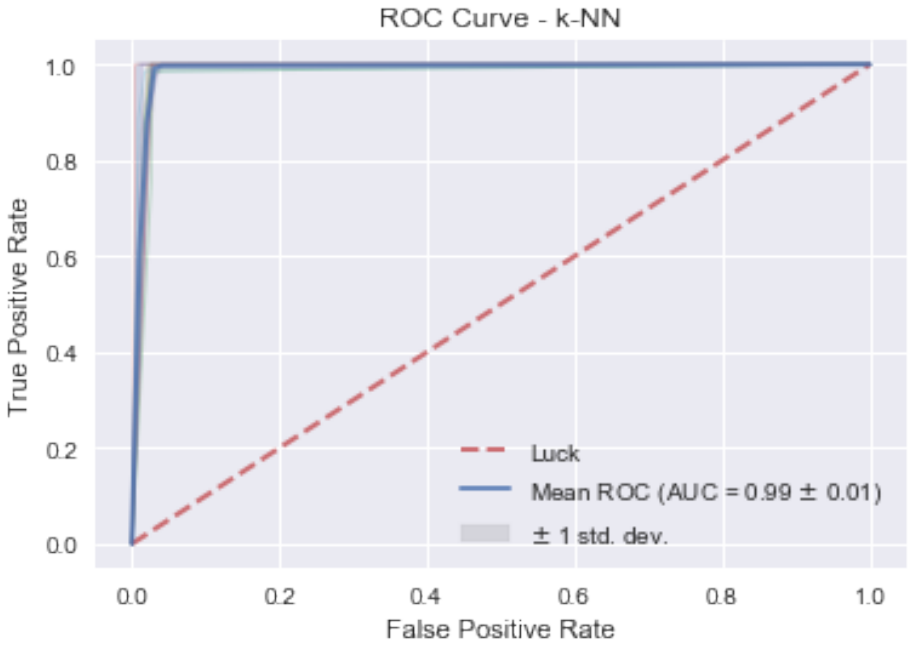


Fig. 4.11 ROC-AUC for k-NN algorithm in the native system

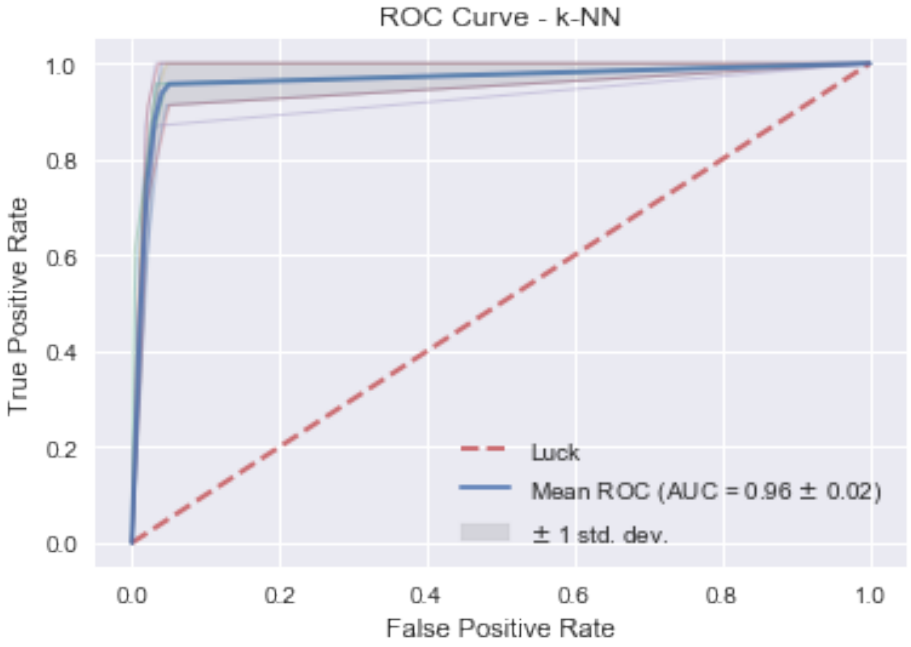


Fig. 4.12 ROC-AUC for k-NN algorithm in the cloud system

events selected, as executing two consecutive instructions produces an equal number of L1, L2 and LLC cache misses. This sets the attack program apart from other workloads as shown in the SPEC benchmark suite which includes two integer applications (bzip2 and gcc) and two floating applications (bwaves and dealII). It is this particularity that enables the k-NN algorithm to build a model identifying the malicious loop in the computational environment with high accuracy. The AMW corroborate the classifier's reliability by slicing the data-set into a sequence of windows of equal size to be searched for phases of the ML. The ML is then seen to be repeating the same task of flushing specific memory address.

The results in Figure 4.11 and 4.12 demonstrate the ability of the k-NN algorithm which builds a classifier that is very accurate in identifying *ML* activities used by the Flush+Reload attack in both native and cloud systems. k-NN is a distance-based algorithm using the search engine to perform classification by finding the closest samples in the data-set. When the *ML* is achieved, the three features (L1, L2 and LLC) have the shortest distance, which is zero in native systems. In cloud systems, on the other hand, the noise in L1 and L2 caches slightly reduces the classifier's accuracy.

Another advantage found in these results is that the profiling can record native and cloud-based activities for the same cost. The same classifier does not require training with different data-sets, and the same data-set can be used to train the classifier to detect malicious processes which are either belong to native or VM process but there will be a 3% degradation in the classifier's accuracy in cloud settings due to the noise.

4.12.5.2 Single Tree C4.5 Results

Figure 4.13 is the ROC metric to evaluate the quality of the single tree classifier in detecting ML activities among normal workloads in the host system. The figure represents the tree classifier's ability to observe program execution attributes and classify the online workloads into malicious and normal activities to measure the risk of existing side channel attack in the system as estimated by ROC-AUC. The model can classify the implicit ML activities inside the Flush+Reload program with accuracy (AUC=0.99 for an average of 10 folds, with a zero confidencezero-confidence interval) in a native system. However, the same algorithm is trained on a data-set which captures the VM activities in a system in which the cloud system is installed in the host OS. The classifier has a lower ability to predict malicious activities among other workloads in the same setting as the native system (AUC=0.89, confidence interval=0.1); the classifier extracts the ML activities with 1% less efficient when identifying the ML activities in the cloud than in native systems. This is because the noise was incurred to L1 and L2 cache memories and the cache misses were increased. This is due to the additional

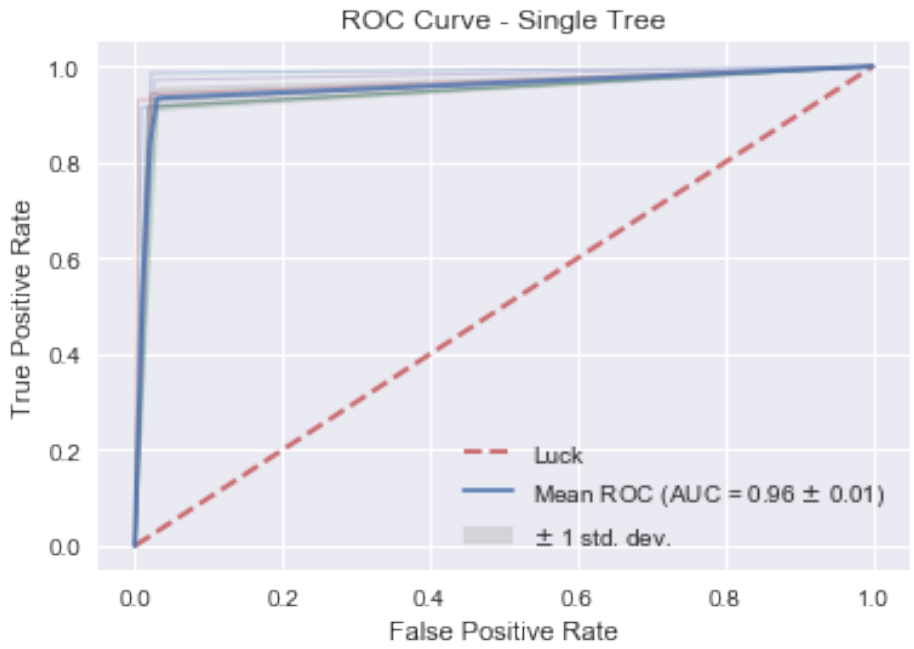


Fig. 4.13 ROC-AUC for single tree algorithm (C4.5) in a native system

translation layer in the Infrastructure as a Service (IaaS) setting in which the hypervisor hides this layer for security reasons across VMs.

The results in 4.13 and 4.14 depict the performance of the C4.5 algorithm in detecting ML activities in the computational environments for both native and cloud workloads. C4.5 grows a tree model to extract the ML activity patterns among various workloads in the system. As can be seen in Figures 4.13 and 4.14, the tree model performance decreases by 1% in the cloud system. This is because C4.5 produces a single tree classifier to predict unseen data points from new data-sets which have failed to contribute relevant features and all samples to support diversity. This leads to both problems of under-fitting and over-fitting. For over-fitting, the model is affected by instability in the execution as mentioned in Section 4.11.3.2. During the side channel program execution, the number of cache misses varies. There might be different ranges of ML execution attributes in training and testing sets. As a result of this, the model can classify very effectively in training but fails for the testing data because the data points in the testing set are not the same as for the training set. Furthermore, in imbalanced data-sets, there is no chance to contribute all data points to the training data-set. However, CV has been used to shuffle the original data-set when the new data-set is created. This is because the model is built on the training set, and new data points might be introduced to the model, and it would be difficult to recognise them.

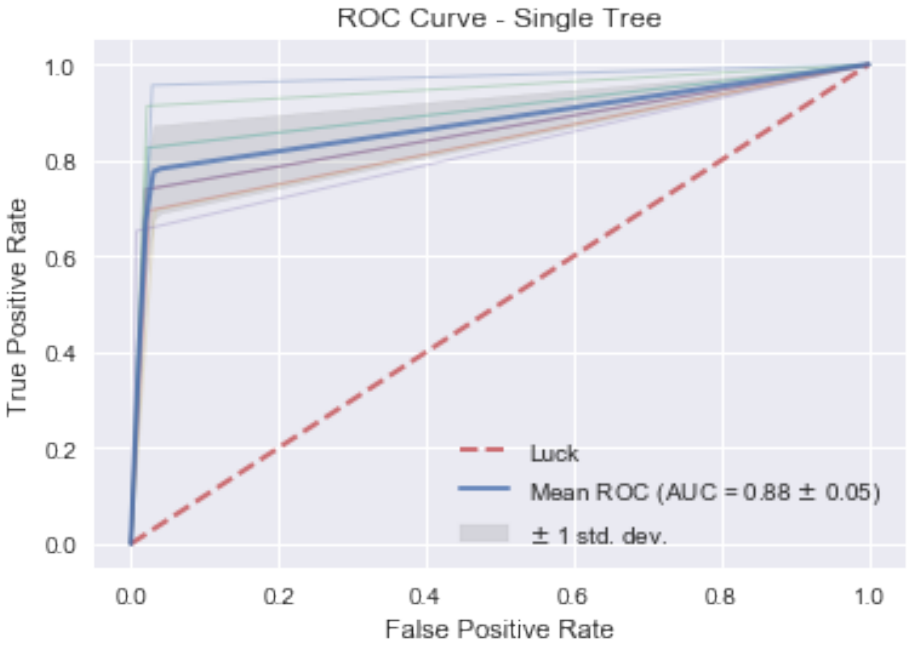


Fig. 4.14 ROC-AUC for single tree algorithm (C4.5) in a cloud system

4.12.5.3 Bagging-Random Forest Results

Figures 4.15 and 4.16 show the ROC metric that evaluates the random-forest classifier's ability to detect the ML activities among normal workloads in the host system in both native and cloud settings respectively. Success in observing program execution attributes and classifying processes as malicious or benign as a measure of the risk of existing side channel attack in the system is shown as estimated by the AUC of ROC. The model identifies ML in a native system with very high accuracy (AUC=0.99 for an average of 10 folds, with a zero confidence interval) 4.15. In the cloud, however, the same algorithm, when trained on a data-set that captured VM activities, was less accurate at predicting malicious activities from among other workloads (AUC=0.99, confidence interval=0.01), as shown in Figure 4.16. The classifier therefore has the same efficiency at identifying malicious loop activities in native and cloud systems. The noise incurred by L1 and L2 cache memories, which arises from the additional translation layer imposed by Infrastructure as a Service (IaaS), has less impact in the random forest model as compared with C4.5 and k-NN. However, the results in Figures 4.12 and 4.14 show that the instability in program execution has a negative impact on the classifier's performance. However, random forest outperformed for single Decision Tree C4.5 and k-NN. This is because random forest utilises a bootstrapping technique 4.12.3 in which all data points in the data-set are involved in model building,

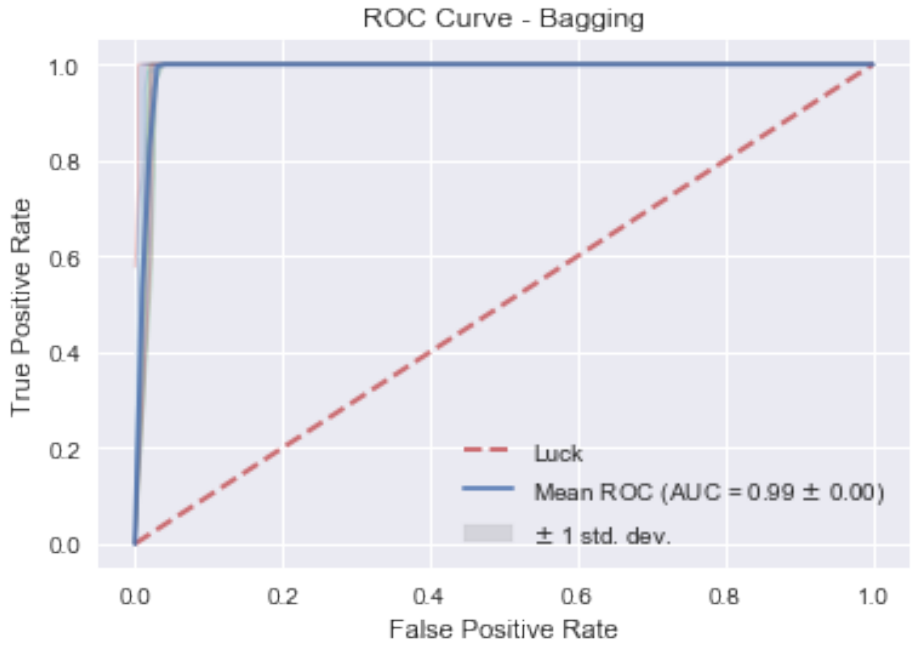


Fig. 4.15 ROC-AUC for bagging algorithm (random forest) in a native system

particularly in the imbalanced data-set. Random forest tries to generate implicit balanced data-sets by bootstrapping the original data-sets; in each data-set, the minor class (attack class) is always placed first, followed by the major class for both training and testing sets, in which case the model is well-trained on the training and testing sets to eliminate under-fitting problems.

4.12.6 Performance

This section reports on the performance overhead which is generated by the detection model. In this experiment, the SPEC CPU2006 benchmark was running for about 13 hours with and without the detection model. The detection model was running in user space and continuously communicating with the Event Record Agent (ERA) to collect data and feed the detection model for malicious activities. Figures 4.17 and 4.18 depict the host OS performance with and without the detection model respectively. The results suggest that the detection model has a very low impact on the performance of the host system; even in the worst case, the performance overhead is within 0.03.

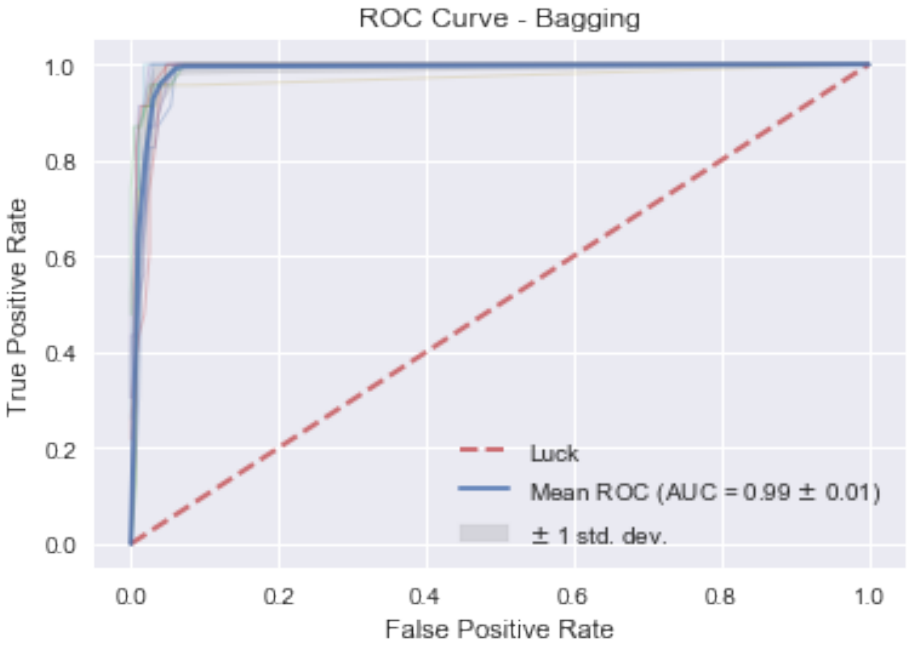


Fig. 4.16 ROC-AUC for bagging algorithm (random forest) in a cloud system

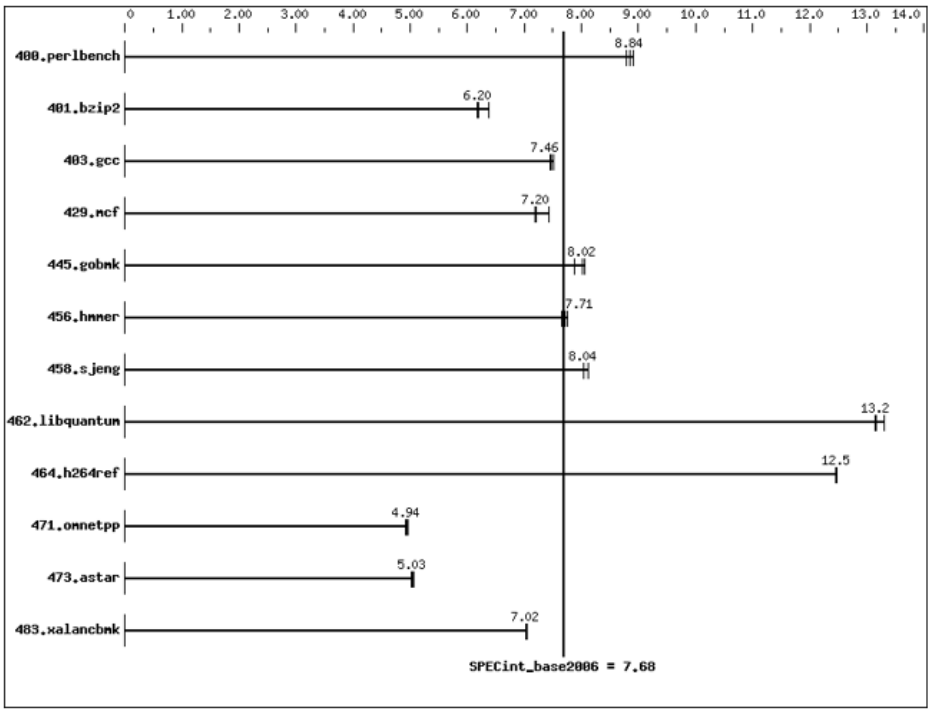


Fig. 4.17 The performance overhead without the detection model using SPEC 2006 benchmark

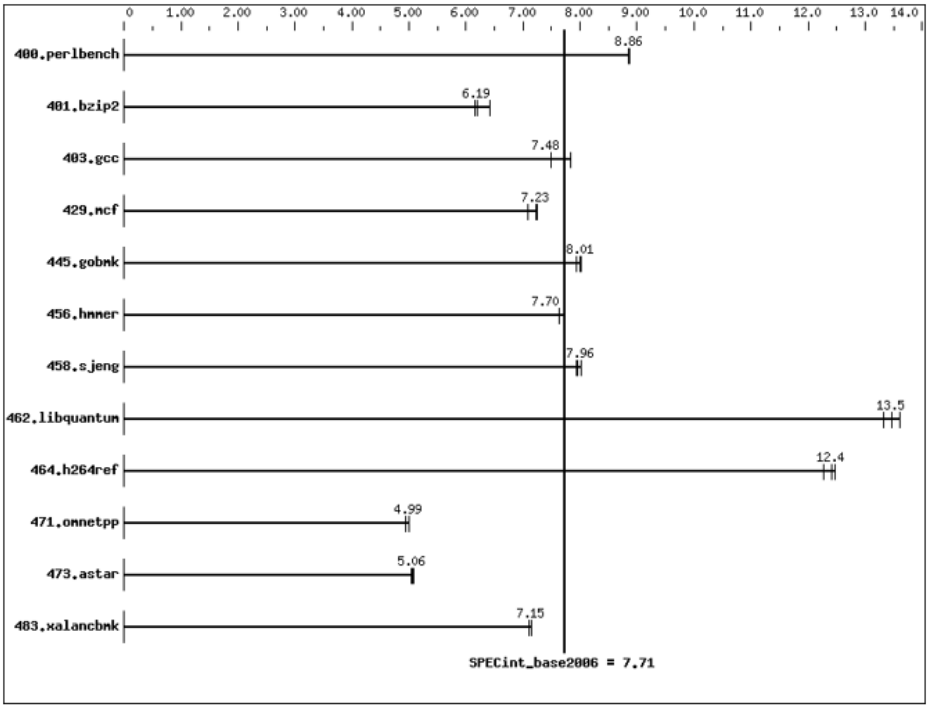


Fig. 4.18 The performance overhead while the detection model is running and SPEC 2006 benchmark

4.12.7 Discussion

In this chapter, we have reported on experiment results reflecting on issues which have an impact on the performance of detection and its robustness in detecting side channel attacks in both native and cloud systems. Our solution is based on three different supervised machine learning techniques and chooses the best algorithm for the detection model. The results suggest that the random forest method can be competitive in detecting side channel attacks. However, feature selection is the core of the detection phase in terms of extracting Flush+Reload attack activities in the computational environment 4.11. This is due to the fact that the detection task is not an easy task due to profiling program execution at the processor core level, in which every single memory transaction will be recorded without knowing the process id PID of the achieved transactions and the duration of the transactions, which means that there is no information about the transactions. Thus, selecting the features which are strongly relevant to the attack activities is essential. Based on the results in Section 4.11, we found that L1, L2 and LLC cache misses have a linear relationship, which supports the detection model to achieve feature extraction with high performance as shown in Section 4.12.5.

In the previous chapter, a synchronisation-based detection of side channel attacks is

conducted by comparing three machine learning algorithms PCANN, k-NN and decision tree algorithms in which decision tree outperformed k-NN and PCANN under various workloads, light and heavy. It can be noticed how the workloads influence the classifier's performance in detecting side channel attacks. The result showed that complex workloads have a negative impact on classifying side channel attacks.

Regarding robustness, the profiling mechanism is not able to recognise the phases of the ML; instead, it is an auto mechanism to capture the program execution attributes. Thus, the MWA algorithm is used to extract the phases of the ML by aggregating the samples of a phase and then moving the entire data-set to inspect any possibilities of ML phases, which are attack activities. Because the processor core profiling is an auto mechanism which does not rely on any means to get information about the processes during their assignment to the processor cores and there is no prior information about processes during their assignment to processor cores. Consequently, the identification mechanism is used to acquire the identity of the attacker. Another benefit of this approach is that the monitoring of program execution activities for native and cloud processes use the same process. This is because native and VM processes are executed concurrently using the same hardware resources (e.g. CPU), unless, in IaaS setting, there is an additional layer in hypervisor which translates virtual addresses into physical addresses. This results to additional noise to the profiling. Thus, the same analysis is used for activities in both native and cloud systems with a slight degradation in the cloud system due to an extra translation layer in hypervisor in cloud systems.

Furthermore, the detection models can identify more than one potential Flush+Reload attack in the system without having any effect on detection accuracy because multiple attacks are independently acting in the system and they never overlap or interfere each other. Thus, they are monitored independently.

Besides this, the identification phase relies on an interrupt and the identification model is executed only if the detection model detects an attack in the system. Consequently, any mis-classification will cause interrupts. The more interrupts, the more significant the performance overhead which is generated in the system. Thus, it is essential that the classification model be sensitive to correctly detect potential attacks.

4.13 Identification Phase

This section describes the necessary background for implementing matching algorithm 6,which demonstrates how the identification model inspects every single memory transaction

in real time and how to change the execution path of the malicious program to interrupt handler when such malicious activities are detected.

4.13.1 Interrupt

An interrupt is an event caused by software interaction with a hardware device. Events may be received as signals by the CPU when they need its attention. The OS will ignore some events, but others cause the OS to handle them immediately; the routine that handles this is called an interrupt handler. Each interrupt handler is designed to respond to a particular event, and falls into one of two categories; hardware interrupts and software interrupts.

1. **Hardware Interrupts:** used to provide communication between such hardware devices as mouse, hard disk, keyboard, and the CPU; purposes may, for example, include advising the operating system of an operation's completion. Alternatively, by sending a signal after reading data from the hard disk, a network interface card tells the CPU about hardware faults. Hardware interrupts are asynchronous and can occur at any time.

   A device is useless without an operating system and kernel. Different types of devices are connected to the central CPU and each device has different characteristics for service provision. A device's performance may be fast or slow, and interrupts enable their use to be properly organised in response to CPU requests.

2. Software interrupts synchronous with program execution, and falling into three types:

   Traps raised by user programs to invoke a system call through the operating system. As an example, a user who wishes to print characters on the screen will cause the program to make system calls on the OS, which then displays the character string and hides the details.

   Exceptions are events generated automatically by the CPU when improperly manipulating instructions. Exceptions are of two types. First, there are Faults (such as page faults) which the CPU can fix ("recover"). Second, Aborts which are irrecoverable by the CPU. Division by zero would be an example. If this exception occurs in the program, the program will terminate because the operating system cannot recover from such a division.

Every CPU in a multi-CPU system has its own dedicated interrupt (INT) pins to receive interrupt signals from external devices. Each hardware device also has a dedicated Interrupt

Request (IRQ). When a device needs to communicate with the CPU, it sends the CPU an interrupt signal through the INT. There are more IRQs in the system than INTs on the CPU, and so the interrupt signals are not transmitted to the CPU immediately but are subject to a Programmable Interrupt Controller (PIC) which organises and prioritises communications requests. The controller sends and responds to requests between specified devices and the CPU. Having been designed for legacy systems, the PIC is limited in the number of devices it can handle and therefore incompatible with multiprocessor systems. Modern multiprocessor systems, with which the PIC was not compatible, instead use Advanced Programmable Interrupt Controller (APIC) which can handle simultaneous multiple interrupt signals across CPU cores. APIC comprises two components: Input/Output APIC (IOAPIC); and Local APIC (LAPIC). Every CPU core has a LAPIC of its own and a motherboard that typically comprises at least one IOAPIC receiving interrupt signals from external devices before distributing them between or across CPU core LAPICs. When an external device such as a keyboard requests an interrupt through IOAPIC, which routes external interrupts to LAPIC and is integrated with LAPICs, it distributes and prioritises the interrupts among CPU cores. They communicate by way of Interrupt Messages and Inter-Process Interrupts (IPI) to distribute interrupts between processors.

The operating system deals with an interrupt either as a masked interrupt or a Non-Maskable Interrupts (NMI). Receipt of a masked interrupt causes the OS to ignore the interrupt, which is in effect disabled. A non-maskable interrupt, on the other hand, demands immediate handling by the OS. A non-maskable interrupt will usually occur only in the event of a critical hardware fault that will cause a system crash. NMI is generally the method of diagnosing such faults.

4.13.2 Identification Model

Figure 4.3 depicts the whole process of detection and identification. From step 5, the Process Identifier Agent (PIA) is the entry point for identification. Let's assume that the identification model has received a message from the detection model. PIA acknowledges the Core Inspector Agent (CIA) to initialise the Trapping procedure. The CIA is a kernel module which is driver-based and where the trapping procedure listens to an incoming message from the PIA. The CIA starts to configure PMCs and initialises parameters including the thresholds, which are obtained by statistical functions in Section 4.11. As it receives the message, the trapping procedure initialises a loop `for_each_online_cpu(cpu)` to examine the online processor cores. This function returns a cpu parameter used by the function `msr`

to confirm that the function reads and it is then pinned to the specific processor core. This is done because, without using `wrmsr_safe_on_cpu(unsigned int cpu, u32 msr_no, u32 l, u32 h)`, there is no guarantee merely from using `rdmsr` and `wrmsr` instructions that the targeted processor core will be read. The CIA then examines each processor core individually to find the core serving the attack program.

Algorithm 6 Identification Algorithm

```
1:  procedure IDENTIFICATION()
2:      threshold, phase, counter
3:      for each core: pc in PC do
4:          obs = read(PMC)
5:          if (obs satisfied thresholds) then
6:              counter++
7:              if counter > phase then
8:                  /* ML is identified, the PMC counters are set to -1 to
9:                  and suspend the MP trigger the interrupt so that
10:                 direct the execution to the interrupt handler */
11:                 modify_PMC(PMC = −1)
12:             end if
13:         end if
14:     end for
15: end procedure
```

A vector of PMC variable $vPMC[pc]$ is created, when $pc =$ the number of PMC, to store PMC counter values. In the inner loop of the identification algorithm 6, in the first iteration, the values of PMCs are stored into $vMPC_0$. In the other iterations, the new captured PMC is averaged with the *vPMC* content. Until the counter reaches the length of the *phase*. The *phase* variable indicates the minimum length of samples which might occur within one job. If, say, the number of samples is five then the loop inside the identification procedure takes five samples and checks the attack pattern by using the threshold parameters discussed in Sections 4.11.2.1; if there is no match between the *vPMC* and the attack thresholds, the loop resets *vPMC* and continues checking. If, on the other hand, the *vPMC* values and the threshold match, this is the process that is causing the attack and the PMC counters are immediately reset to -1, which is explained in details in Section 3.2.3, to force a PMC overflow using the current process core. Recall, PMC interrupt is enabled (see Section 4.13.1). The PMC counter overflow causes the OS to suspend the current process assigned to the current processor core and hands control to the Trapping interrupt handler. Inside the Trapping interrupt handler, information about the suspended processor core is taken from the

Processor Control Block (PCB) and passed back to PIA, which will now have the identity of the malicious process and can take necessary action to find its owner.

4.13.3 Identification Phase Evaluation

This section discusses and evaluates the results obtained from the experiments. To evaluate the identification, three experiments are conducted in which the only difference is the profiling settings. What is changed is the number of samples {5, 10, 20} taken by the trapping task. The number of samples is critical to identification, because more than the threshold causes the identification model to miss the attacker; or less than the threshold for the identification model leads to the generation of a False Negative (FN) (detecting normal activities as attack activities) due to inferring non-attack samples.

Figure 4.19 shows profiling with 1000 samples for each native and cloud Flush+Reload program. The duration of Flush+Reload program jobs running in native and cloud are different. The time quantum for the cloud-based jobs is longer than for the native ones. This duration has an impact on identifying the malicious process activities, which are denoted in the red and blue horizontal lines. Green boundaries show correct detection of the attack program by the algorithm; red boundaries show a failure to capture the attack program. The boundaries are not equal due to the soft scheduler, in which there is flexibility for the jobs to be completed. By relying on the analysis in Section 4.11.2.1, we can define the minimum and maximum required time to complete a job. Figure 4.19 shows the difference between the quanta for native and cloud jobs. Sub-figure (a) shows the scheduler for real-time executions in user space for a native system. Sub-figure (b) shows the scheduler for real-time execution of VM and host real time programs in user space. The VM job has a larger quantum than the native-based jobs. This is because recent work shows that the time quantum for jobs for VM processes is longer than for jobs in a native system for performance purposes. Thus, the identification model has more confidence in detecting malicious VM than a native-based malicious program.

The best attack scenarios for a native Flush+Reload attack are when the identification model starts at the same time as the jobs of the ML are assigned to the processor core and start executing. In less likely attack scenarios, the identification model starts in the middle of the jobs and, in this case, the activities of other workloads will interfere with the observations and fail in matching.

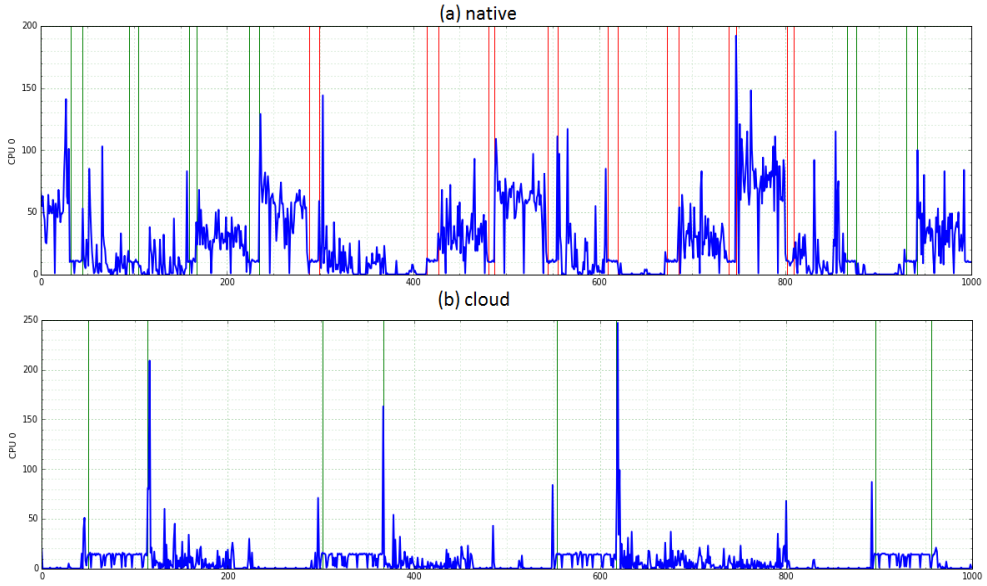


Fig. 4.19 The execution time line which is sliced among the attacker in a native system and 4 SPEC workloads. The malicious loop inside Flush+Reload program phases for LLC cache misses appear as chunks of samples which can be observed as process transactions on a specific processor core.

4.14 Discussion

None of the detection and prevention techniques so far developed can prevent a side channel attack from taking place. They can make attacks more difficult to carry out, but they cannot stop attackers from achieving their malicious aims, because vulnerabilities in hardware and software allow attackers to find ways round them. Ge et al. (2017) concluded that the most commonly used current processors, including x86 and ARM processors, are not designed in such a way as to maintain critical security for the core-processor. lipp2018meltdown suggests to microprocessor manufacturers that performance should not be the sole aim of chip design and that they should be more concerned with making communication channels in the computational environment completely secure.

What stands in the way of side channel attack mitigation is the overheads generated by protection methods and the fact that implementing those methods is complex. The overheads in question relate to the fact that operating systems operate on one layer and applications on another. OS overheads negatively impact on the system, including user programs running on it, but overheads generated in the application levels affect the targeted applications rather than the system and its complexity.

The fact is, though, that the majority of proposed side channel attacks and countermea-

sures against them are reliant on assumptions, the enabling and disabling features being an example. It follows that building a reliable security model requires study of those vulnerabilities that attackers exploit and of the limitations in the ability of existing countermeasures to provide security tailored to specific situations.

The classification problem requires more details to improve the solution by addressing every single mistake made by the model while classifying unseen data. In this study, classification is the framework's core task, because 4.12.2 after detecting an attack, the Detection Agent (DA) sends a message to the Process Identification Agent (PIA) to identify the attacker. The identification phase relies on an interrupt, so that any mis-classification will cause interrupts, which leads to the generation of performance overheads. The more interrupts there are, the more significant are performance overheads generated in the system. It is therefore essential that the classification model be sensitive for correct detection of an attack.

As was mentioned in Section 4.5, most attackers are not aware of the organised (repeatable) unintentional contentions. For instance, if (Del Pozo et al., 2015) tries to evade the detection system it is true that a lot of noise will be incurred in the observations; on the other hand, the detection system is sensible at a degree even if the attacker tries to deviate the detection system by slowing down the scanning mechanism.

The profiling in this chapter is based on the underlying processor core for which it is important to know the behaviour or activities of the jobs and the duration of their assignment to the processor cores. The details are given in Section 4.10. Consequently, program phase detection mechanisms have been used to efficiently identify the ML loop repetitions and apply the sum of aggregation function to bound the ML's execution attributes in one data point in the data-set before feeding them to the classification algorithms. Consequently, the program phase supports the classifier to be more reliable and robust in detecting ML iterations. In this case, the performance overheads of the system are reduced, because fewer interrupts are triggered.

As the experiment results show, identifying the Flush+Reload program is more guaranteed while it is running in a cloud rather than a native system. This is because recent work has shown that the time quantum for jobs of VM processes are longer than for jobs in a native system. Short time slicing in cloud systems will lead to degradation in the performance of the system. This will lead to a better chance to monitor malicious VMs with high confidence to make decisions about detecting and identifying the attacker.

Chapter 5

Conclusions And Future Work

This Chapter presents the conclusions drawn from the utilised methods of the detection system in real-time systems and the potential direction of future work are discussed.

5.1 Conclusions

This section summarises and discusses the research contributions of the thesis. Further, the key design of the framework are highlighted in away that they shape the scope of the work.

5.1.1 Research Summary

This work addressed the problem of perplexing approaches in detecting side channel attack and identifying the attackers in both native and cloud systems by using machine learning approaches. This problem results in difficulties in confining malicious processes to identify the attacker. This is particularly in the cloud system in which it is crucial to identify VMs which achieve side channel attacks. The hypothesis stimulating this work is based on the possibility to model and analyse complex computational environments concerning user-defined security in an automated manner.

In this thesis a new framework was developed for system security, which allows operators, or cloud providers to implement security mechanisms for their consumers. Based on the improvement of the existing detection systems, the objectives of the proposed framework are to secure the computational environments from side channel attacks and identify the owner of the attacker, initiated by Flush+Reload scheme.

The framework is developed on the ground of machine learning algorithms which is

capable of efficiently detecting side channel attack. This supports identification phase to disrupt the attack before stealing the entire key bits of secret elements.

5.1.2 Contribution to knowledge

The main finding of this thesis is that machine learning algorithms can mitigate side channel attacks in host systems. This has been achieved by developing a real-time accurate and fast prediction system. Previous detection systems relied on the synchronisation between the victims and attackers' programs, running concurrently. However, this thesis demonstrates that the approach of using direct profiling processor cores for any memory transactions had clear advantages over synchronisation approach. The benefit of the approach, presented in this thesis, is the ability of detecting all potential Flush+Reload programs in the system as they are running concurrently in the system.

A profiling mechanism was suggested which is capable of monitoring program execution attributes at processor cores level to capture the execution of malicious programs by deploying program phase detection techniques. The statistical analysis introduced to determine the OS's scheduler behaviour rely on its uniqueness while detecting both malicious VMs and native processes at no extra cost to the performance. The uniqueness of the attacker program was analysed by utilising Descriptive Statistics. Various experiments were conducted to capture the attacker behaviour in different scenarios to generalise the detection and identification of such attacks. The profiling mechanism was also able to filter out the system workloads' noises. In addition, the program execution instability was addressed through different scenarios which helped the identification phase by providing thresholds and possibilities of the program executions in different runs.

The major contribution of this thesis was to bring the program phase detection into detection system by identifying the fine-grained program execution attributes of Malicious Loop inside Flush+Reload program in real-time systems. In the processor core profiling level, it is challenging to identity the malicious processes through the phase detection; the owner of the attack can be identified without sense of the attacker. Moreover, this thesis demonstrated the benefit of profiling mechanism to construct the inner loop of ML inside Flush+Reload program while the processes of the program are fragmented into a significant number of jobs. Besides, the framework of this thesis benefits from very low overhead performance approximately less than 1% of the host system.

The source of information, which is the core of the detection system, to investigate the presence of side channel attack was addressed in this research. The initial analysis showed the

feasibility of utilising machine learning methods, by comparing the result of three algorithms to detect side channel attacks with various workloads using SPEC CPU 2006 benchmark suite. Then the tree algorithm was selected which outperformed the other algorithms in this environment (PCANN and k-NN). We also explored a range of machine learning techniques and addressed the limitations of single decision tree algorithms in the context of our work. The results showed that Random forest, which is the implementation of bagging technique, has better efficiency than single tree algorithms. The model could significantly be optimised by utilising bagging technique.

5.2 Limitations

This section describes the limitations of the proposed framework in this research including both detection and identification phases.

1. The framework presented within this thesis does not only rely on the learning detection models, but also, in case the detection model fails to detect, the detection system introduces extra overhead to the system.

2. The framework proposed in this research cannot be generalised for similar attack such as Prime+Probe. Prime+Probe requires different features to be extracted in time execution line. However, the same analysis mechanism can be used for other side channel attack techniques.

3. Attackers can use different processes to achieve the attack. For instance, if Flush+Reload task is carried out to one process and the rest of the attack tasks on different processes, then the framework will fail to detect the other processes related to the attacker. However, by identifying the malicious VMs, another action can take place to detect all processes belong to the VMs.

5.3 Future Work

Side channel attacks primarily utilise Flush+Reload and Prime+Probe techniques to exploit hardware and software vulnerabilities. Current work has focused on Flush+Reload attack. In the future work, we will extend the framework to detect Prime+Probe technique in real-time systems. Moreover, we believe that hardware-based security rely on HPC has not yet been explored well. Based on our findings, HPC can be utilised to monitor and extract a small

amount of code inside a program precisely. Consequently, HPC can be used to trace attacks such as Meltdown and Spectre in which multiple techniques have been utilised to achieve the attack.

References

Aciiçmez, O. (2007). Yet another microarchitectural attack:: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18. ACM.

Acıiçmez, O., Brumley, B. B., and Grabher, P. (2010). New results on instruction cache attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 110–124. Springer.

Acıiçmez, O., Koç, Ç. K., and Seifert, J.-P. (2007). Predicting secret keys via branch prediction. In *CT-RSA*, volume 2007, pages 225–242. Springer.

Aciicmez, O. and Seifert, J.-P. (2007). Cheap hardware parallelism implies cheap security. In *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*, pages 80–91. IEEE.

Alam, M., Bhattacharya, S., Mukhopadhyay, D., and Bhattacharya, S. (2017). Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks. https://eprint.iacr.org/2017/564.

Allaf, Z., Adda, M., and Gegov, A. (2017). A comparison study on flush+ reload and prime+ probe attacks on aes using machine learning approaches. In *UK Workshop on Computational Intelligence*, pages 203–213. Springer.

Allaf, Z., Adda, M., and Gegov, A. (2018). Confmvm: A hardware-assisted model to confine malicious vms. In *UKSim2018: UKSim-AMSS 20th International Conference on Modelling & Simulation*. IEEE.

Aly, H. and ElGayyar, M. (2013). Attacking aes using bernstein's attack on modern processors. In *Progress in Cryptology–AFRICACRYPT 2013*, pages 127–139. Springer.

Arcangeli, A., Eidus, I., and Wright, C. (2009). Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28. Citeseer.

Arlot, S., Celisse, A., et al. (2010). A survey of cross-validation procedures for model selection. *Statistics surveys*, 4:40–79.

Backes, M., Dürmuth, M., Gerling, S., Pinkal, M., and Sporleder, C. (2010). Acoustic side-channel attacks on printers. In *USENIX Security symposium*, pages 307–322.

Bala, V., Duesterwald, E., and Banerjia, S. (2011). Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 46(4):41–52.

Banerjee, U., Ho, L., and Koppula, S. (2015). Power-based side-channel attack for aes key extraction on the atmega328 microcontroller.

Barandiaran, I. (1998). The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8).

Bernstein, D. J. (2005). Cache-timing attacks on aes.

Bernstein, D. J., Breitner, J., Genkin, D., Bruinderink, L. G., Heninger, N., Lange, T., van Vredendaal, C., and Yarom, Y. (2017). Sliding right into disaster: Left-to-right sliding windows leak. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 555–576. Springer.

Bosman, E., Razavi, K., Bos, H., and Giuffrida, C. (2016). Dedup est machina: Memory deduplication as an advanced exploitation vector.

Bradley, A. P. (1997). The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159.

Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. (2017). Software grand exposure: Sgx cache attacks are practical. *arXiv preprint arXiv:1702.07521*.

Breiman, L. (1996a). Bagging predictors. *Machine learning*, 24(2):123–140.

Breiman, L. (1996b). Bias, variance, and arcing classifiers.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Brickell, E., Graunke, G., Neve, M., and Seifert, J.-P. (2006). Software mitigations to hedge aes against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52.

Briongos, S., Irazoqui, G., Malagón, P., and Eisenbarth, T. (2017). Cacheshield: Protecting legacy processes against cache attacks. *arXiv preprint arXiv:1709.01795*.

Briongos, S., Malagón, P., Risco-Martín, J. L., and Moya, J. M. (2016). Modeling side-channel cache attacks on aes. In *Proceedings of the Summer Computer Simulation Conference*, page 37. Society for Computer Simulation International.

Bruinderink, L. G., Hülsing, A., Lange, T., and Yarom, Y. (2016). Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 323–345. Springer.

Cai, L. and Chen, H. (2011). Touchlogger: Inferring keystrokes on touch screen from smartphone motion. *HotSec*, 11:9–9.

Cauligi, S., Soeller, G., Brown, F., Johannesmeyer, B., Huang, Y., Jhala, R., and Stefan, D. (2017). Fact: A flexible, constant-time programming language. In *Cybersecurity Development (SecDev), 2017 IEEE*, pages 69–76. IEEE.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357.

Chen, S., Wang, R., Wang, X., and Zhang, K. (2010). Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *2010 IEEE Symposium on Security and Privacy*, pages 191–206. IEEE.

Chiappetta, M., Savas, E., and Yilmaz, C. (2015). Real time detection of cache-based side-channel attacks using hardware performance counters. Technical report, Cryptology ePrint Archive, Report 2015/1034.

Chu, C.-K., Zhu, W.-T., Han, J., Liu, J. K., JIA, X., and JIANYING, Z. (2013). Security concerns in popular cloud storage services. *IEEE pervasive computing*, 12(4):50–57.

Cieslak, D. A. and Chawla, N. V. (2008). Learning decision trees for unbalanced data. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 241–256. Springer.

Cleemput, J. V., Coppens, B., and De Sutter, B. (2012). Compiler mitigations for time attacks on modern x86 processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):23.

Coppens, B., Verbauwhede, I., De Bosschere, K., and De Sutter, B. (2009). Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 45–60. IEEE.

Costan, V., Lebedev, I. A., and Devadas, S. (2016). Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874.

Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27.

Crane, S., Homescu, A., Brunthaler, S., Larsen, P., and Franz, M. (2015). Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, pages 8–11.

Dachman-Soled, D., Locke, S. N., Maimon, S., Metzger, R., Shahverdi, A., and Sullivan-Russett, L. B. (2017). Side-channel attacks on btree.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. (2012). Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231.

Del Pozo, S. M., Standaert, F.-X., Kamel, D., and Moradi, A. (2015). Side-channel attacks from static power: When should we care? In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 145–150. EDA Consortium.

Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., and Stolfo, S. (2013). On the feasibility of online malware detection with performance counters. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 559–570. ACM.

Dhodapkar, A. S. and Smith, J. E. (2003). Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 217. IEEE Computer Society.

Ding, C., Dwarkadas, S., Huang, M. C., Shen, K., and Carter, J. B. (2006). Program phase detection and exploitation. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE.

Dongarra, J., London, K., Moore, S., Mucci, P., and Terpstra, D. (2001). Using papi for hardware performance monitoring on linux systems. In *Conference on Linux Clusters: The HPC Revolution*, volume 5. Linux Clusters Institute.

Ducas, L. (2014). Accelerating bliss: the geometry of ternary polynomials. *IACR Cryptology ePrint Archive*, 2014:874.

Eijkhout, V. (2015). *Introduction to High Performance Scientific Computing*.

Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. (2016). Jump over aslr: Attacking branch predictors to bypass aslr. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE.

Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. (2006). A performance counter architecture for computing accurate cpi components. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 175–184. ACM.

Fawcett, T. (2006). An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874.

Fei, Y., Ding, A. A., Lao, J., and Zhang, L. (2014). A statistics-based fundamental model for side-channel attack analysis. *IACR Cryptology ePrint Archive*, 2014:152.

Galar, M., Fernandez, A., Barrenechea, E., Bustince, H., and Herrera, F. (2012). A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(4):463–484.

Ge, Q., Yarom, Y., Cock, D., and Heiser, G. (2016). A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, pages 1–27.

Ge, Q., Yarom, Y., and HEISER, G. (2017). Your processor leaks information-and there's nothing you can do about it. *CoRR abs/1612.04474*.

Genkin, D., Valenta, L., and Yarom, Y. (2017). May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 845–858.

Geurts, P. and Louppe, G. (2011). Learning to rank with extremely randomized trees. In *JMLR: Workshop and Conference Proceedings*, volume 14, pages 49–61.

Gruss, D., Bidner, D., and Mangard, S. (2015a). Practical memory deduplication attacks in sandboxed javascript. In *European Symposium on Research in Computer Security*, pages 108–122. Springer.

Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. (2017a). Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer.

Gruss, D., Lipp, M., Schwarz, M., Genkin, D., Juffinger, J., O'Connell, S., Schoechl, W., and Yarom, Y. (2017b). Another flip in the wall of rowhammer defenses. *arXiv preprint arXiv:1710.00551*.

Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. (2016a). Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379. ACM.

Gruss, D., Maurice, C., and Mangard, S. (2016b). Rowhammer. js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer.

Gruss, D., Maurice, C., and Wagner, K. (2015b). Flush+ flush: A stealthier last-level cache attack. *arXiv preprint arXiv:1511.04594*.

Gruss, D., Maurice, C., Wagner, K., and Mangard, S. (2015c). Flush+ flush: A fast and stealthy cache attack. *arXiv preprint arXiv:1511.04594*.

Gruss, D., Spreitzer, R., and Mangard, S. (2015d). Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912.

Gueron, S. (2008). Advanced encryption standard (aes) instructions set. *Intel, http://softwarecommunity. intel. com/articles/eng/3788. htm, accessed*, 25.

Gullasch, D., Bangerter, E., and Krenn, S. (2011). Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE.

Gulmezoglu, B., Eisenbarth, T., and Sunar, B. (2017). Cache-based application detection in the cloud using machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 288–300. ACM.

Gulmezoglu, B., Inci, M., Irazoki, G., Eisenbarth, T., and Sunar, B. (2016). Cross-vm cache attacks on aes.

Gülmezoğlu, B., Inci, M. S., Irazoqui, G., Eisenbarth, T., and Sunar, B. (2015). A faster and more realistic flush+ reload attack on aes. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 111–126. Springer.

Gupta, A. (2017). *Assessing Hardware Performance Counters for Malware Detection*. PhD thesis, Boston University.

Hamburg, M. (2009). Accelerating aes with vector permute instructions. In *CHES*, volume 5747, pages 18–32. Springer.

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17.

Hernandez-Castro, C. J. and Ribagorda, A. (2010). Pitfalls in captcha design and implementation: The math captcha, a case study. *computers & security*, 29(1):141–157.

Hovhannisyan, H., Lu, K., and Wang, J. (2015). A novel high-speed ip-timing covert channel: Design and evaluation. In *2015 IEEE International Conference on Communications (ICC)*, pages 7198–7203. IEEE.

Hund, R., Willems, C., and Holz, T. (2013). Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE.

Hunt, E. B., Marin, J., and Stone, P. J. (1966). Experiments in induction.

Intel, I. (2014). Software guard extensions programming reference, revision 2.

Irazoqui, G., Eisenbarth, T., and Sunar, B. (2015). S $ a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604. IEEE.

Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. (2014). Wait a minute! a fast, cross-vm attack on aes. In *International Workshop on Recent Advances in Intrusion Detection*, pages 299–319. Springer.

Jana, S. and Shmatikov, V. (2012). Memento: Learning secrets from process footprints. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 143–157. IEEE.

Japkowicz, N. (2000). The class imbalance problem: Significance and strategies. In *Proc. of the Int'l Conf. on Artificial Intelligence*.

Johnson, M., McCraw, H., Moore, S., Mucci, P. J., Nelson, J., Terpstra, D., Weaver, V. M., and Mohan, T. (2012). Papi-v: Performance monitoring for virtual machines. In *ICPP Workshops*, pages 194–199.

Käsper, E. and Schwabe, P. (2009). Faster and timing-attack resistant aes-gcm. In *CHES*, volume 5747, pages 1–17. Springer.

Kayaalp, M., Abu-Ghazaleh, N., Ponomarev, D., and Jaleel, A. (2016). A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*, page 72. ACM.

Kayaalp, M., Khasawneh, K. N., Esfeden, H. A., Elwell, J., Abu-Ghazaleh, N., Ponomarev, D., and Jaleel, A. (2017). Ric: Relaxed inclusion caches for mitigating llc side-channel attacks. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 7. ACM.

Kellaris, G., Kollios, G., Nissim, K., and O'Neill, A. (2016). Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1340. ACM.

Kim, T., Peinado, M., and Mainar-Ruiz, G. (2012). Stealthmem: system-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kirovski, D., Drinić, M., and Potkonjak, M. (2002). Enabling trusted software integrity. In *ACM SIGPLAN Notices*, volume 37, pages 108–120. ACM.

Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2018). Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*.

Kocher, P., Jaffe, J., Jun, B., and Rohatgi, P. (2011). Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27.

Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer.

Kohavi, R. et al. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada.

Kulah, Y., Dincer, B., Yilmaz, C., and Savas, E. (2018). Spydetector: An approach for detecting side-channel attacks at runtime. *International Journal of Information Security*, pages 1–30.

Lampson, B. W. (1969). Dynamic protection structures. In *Proceedings of the November 18-20, 1969, fall joint computer conference*, pages 27–38. ACM.

Lau, J., Schoenmackers, S., and Calder, B. (2005). Transition phase classification and prediction. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 278–289. IEEE.

Li, C. (2007). Classifying imbalanced data using a bagging ensemble variation (bev). In *Proceedings of the 45th annual southeast regional conference*, pages 203–208. ACM.

Lipp, M., Gruss, D., Schwarz, M., Bidner, D., Maurice, C., and Mangard, S. (2017). Practical keystroke timing attacks in sandboxed javascript. In *European Symposium on Research in Computer Security*, pages 191–209. Springer.

Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. (2016). Armageddon: Cache attacks on mobile devices. In *USENIX Security Symposium*, pages 549–564.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown. *arXiv preprint arXiv:1801.01207*.

Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., and Lee, R. B. (2016). Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418. IEEE.

Malone, C., Zahran, M., and Karri, R. (2011). Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the sixth ACM workshop on Scalable trusted computing*, pages 71–76. ACM.

Marshall, A., Howard, M., Bugher, G., Harden, B., Kaufman, C., Rues, M., and Bertocci, V. (2010). Security best practices for developing windows azure applications. *Microsoft Corp*, page 1.

Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., and Francillon, A. (2015a). Reverse engineering intel last-level cache complex addressing using performance counters. In *International Workshop on Recent Advances in Intrusion Detection*, pages 48–65. Springer.

Maurice, C., Neumann, C., Heen, O., and Francillon, A. (2015b). C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64. Springer.

Mell, P. and Grance, T. (2011). The nist definition of cloud computing.

Messerges, T. S., Dabbish, E. A., and Sloan, R. H. (1999). Investigations of power analysis attacks on smartcards. *Smartcard*, 99:151–161.

Messerges, T. S., Dabbish, E. A., and Sloan, R. H. (2002). Examining smart-card security under the threat of power analysis attacks. *IEEE transactions on computers*, 51(5):541–552.

Moghimi, A., Irazoqui, G., and Eisenbarth, T. (2017). Cachezoom: How sgx amplifies the power of cache attacks. *arXiv preprint arXiv:1703.06986*.

Neve, M., Seifert, J.-P., and Wang, Z. (2006). A refined look at bernstein's aes side-channel analysis. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 369–369. ACM.

Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., Le, Q. V., and Ng, A. Y. (2011). On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 265–272.

Nikulin, V., McLachlan, G. J., and Ng, S. K. (2009). Ensemble approach for the classification of imbalanced data. In *Australasian Joint Conference on Artificial Intelligence*, pages 291–300. Springer.

Nomani, J. and Szefer, J. (2015). Predicting program phases and defending against side-channel attacks using hardware performance counters. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, page 9. ACM.

of Technology, G. U. (2018). Meltdown and spectre attacks.

Osvik, D. A., Shamir, A., and Tromer, E. (2006). Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology–CT-RSA 2006*, pages 1–20. Springer.

Page, D. (2003). Defending against cache-based side-channel attacks. *Information Security Technical Report*, 8(1):30–44.

Pan, Y.-S., Chiang, J.-H., Li, H.-L., Tsao, P.-J., Lin, M.-F., and Chiueh, T.-c. (2011). Hypervisor support for efficient memory de-duplication. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 33–39. IEEE.

Paoloni, G. (2010). How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. *Intel Corporation, September*, 123.

Payer, M. (2016). Hexpads: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*, pages 138–154. Springer.

Penchalaiah, N. and Seshadri, R. (2010). Effective comparison and evaluation of des and rijndael algorithm (aes). *International Journal of Computer Science and Engineering*, 2(05):1641–1645.

Percival, C. (2005). Cache missing for fun and profit.

Pessl, P., Bruinderink, L. G., and Yarom, Y. (2017). To bliss-b or not to be-attacking strongswan's implementation of post-quantum signatures.

Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. (2016). Drama: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security Symposium*, pages 565–581.

Pfoh, J., Schneider, C., and Eckert, C. (2011). Nitro: Hardware-based system call tracing for virtual machines. *Advances in information and computer security*, pages 96–112.

Picek, S., Heuser, A., Jovic, A., and Legay, A. (2017). Climbing down the hierarchy: Hierarchical classification for machine learning side-channel attacks. In *International Conference on Cryptology in Africa*, pages 61–78. Springer.

Pornin, T. (2016). Bearssl: A smaller ssl/tls library.

Qian, Z., Mao, Z. M., and Xie, Y. (2012). Collaborative tcp sequence number inference attack: how to crack sequence number under a second. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 593–604. ACM.

Reparaz, O., Balasch, J., and Verbauwhede, I. (2017). Dude, is my code constant time? In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1697–1702. IEEE.

Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. (2009). Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM.

Rokach, L. (2016). Decision forest: Twenty years of research. *Information Fusion*, 27:111–125.

Sandberg, A., Sembrant, A., Hagersten, E., and Black-Schaffer, D. (2013). Modeling performance variation due to cache sharing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 155–166. IEEE.

Santhanam, G. R., Holland, B., Kothari, S., and Ranade, N. (2017). Human-on-the-loop automation for detecting software side-channel vulnerabilities. In *International Conference on Information Systems Security*, pages 209–230. Springer.

Sayadi, H., Patel, N., PD, S. M., Sasan, A., Rafatirad, S., and Homayoun, H. (2018). Ensemble learning for effective run-time hardware-based malware detection: A comprehensive analysis and classification. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE.

Schwarz, M., Gruss, D., Lipp, M., Maurice, C., Schuster, T., Fogh, A., and Mangard, S. (2017a). Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. *arXiv preprint arXiv:1711.01254*.

Schwarz, M., Lipp, M., Gruss, D., Weiser, S., Maurice, C., Spreitzer, R., and Mangard, S. (2017b). Keydrown: Eliminating keystroke timing side-channel attacks. *arXiv preprint arXiv:1706.06381*.

Seaborn, M. and Dullien, T. (2015). Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*.

Shabtai, A., Elovici, Y., and Rokach, L. (2012). *A survey of data leakage detection and prevention solutions*. Springer Science & Business Media.

Shafer, J., Agrawal, R., and Mehta, M. (1996). Sprint: A scalable parallel classi er for data mining. In *Proc. 1996 Int. Conf. Very Large Data Bases*, pages 544–555. Citeseer.

Shah, G. and Blaze, M. (2009). Covert channels through external interference. In *Proceedings of the 3rd USENIX conference on Offensive technologies (WOOT'09)*, pages 1–7.

Shah, G., Molina, A., Blaze, M., et al. (2006). Keyboards and covert channels. In *Usenix security*, volume 6, pages 59–75.

Sherwood, T., Perelman, E., Hamerly, G., Sair, S., and Calder, B. (2003a). Discovering and exploiting program phases. *IEEE micro*, 23(6):84–93.

Sherwood, T., Sair, S., and Calder, B. (2003b). Phase tracking and prediction. In *ACM SIGARCH Computer Architecture News*, volume 31, pages 336–349. ACM.

Shinde, S., Chua, Z. L., Narayanan, V., and Saxena, P. (2016). Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328. ACM.

Simakov, N. A., Innus, M. D., Jones, M. D., White, J. P., Gallo, S. M., DeLeon, R. L., and Furlani, T. R. (2018). Effect of meltdown and spectre patches on the performance of hpc applications. *arXiv preprint arXiv:1801.04329*.

Skrenes, A. and Williamson, C. (2016). Experimental calibration and validation of a speed scaling simulator. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*, pages 105–114. IEEE.

Song, D. X., Wagner, D., and Tian, X. (2001). Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium*, volume 2001.

Stolfo, S. J., Salem, M. B., and Keromytis, A. D. (2012). Fog computing: Mitigating insider data theft attacks in the cloud. In *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*, pages 125–128. IEEE.

Suzaki, K., Iijima, K., Yagi, T., and Artho, C. (2011). Memory deduplication as a threat to the guest os. In *Proceedings of the Fourth European Workshop on System Security*, page 1. ACM.

Tang, A., Sethumadhavan, S., and Stolfo, S. J. (2014). Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection*, pages 109–129. Springer.

Terpstra, D., Jagode, H., You, H., and Dongarra, J. (2010). Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer.

Tromer, E., Osvik, D. A., and Shamir, A. (2010). Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71.

Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., and Miyauchi, H. (2003). Cryptanalysis of des implemented on computers with cache. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 62–76. Springer.

Valentini, G. L., Lassonde, W., Khan, S. U., Min-Allah, N., Madani, S. A., Li, J., Zhang, L., Wang, L., Ghani, N., Kolodziej, J., et al. (2013). An overview of energy efficiency techniques in cluster computing systems. *Cluster Computing*, 16(1):3–15.

van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., and Giuffrida, C. (2016). Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689. ACM.

Varadarajan, V., Ristenpart, T., and Swift, M. M. (2014). Scheduler-based defenses against cross-vm side-channels. In *USENIX Security Symposium*, pages 687–702.

VMWARE, I. (2009). Understanding memory resource management in vmware esx server. *Palo Alto, California, United States*.

Vogl, S. and Eckert, C. (2012). Using hardware performance events for instruction-level monitoring on the x86 architecture. In *Proceedings of the 2012 European Workshop on System Security EuroSec*, volume 12.

Waldspurger, C. A. (2002). Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194.

Wang, C., Wang, Q., Ren, K., Cao, N., and Lou, W. (2012). Toward secure and dependable storage services in cloud computing. *Services Computing, IEEE Transactions on*, 5(2):220–232.

Wang, X. and Karri, R. (2013). Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *Proceedings of the 50th Annual Design Automation Conference*, page 79. ACM.

Wang, Z. and Lee, R. B. (2006). Covert and side channels due to processor architecture. In *ACSAC*, volume 6, pages 473–482.

Wu, J., Kim, Y.-B., and Choi, M. (2010). Low-power side-channel attack-resistant asynchronous s-box design for aes cryptosystems. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, pages 459–464. ACM.

Wu, Z., Xu, Z., and Wang, H. (2012). Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 159–173.

Wu, Z., Xu, Z., and Wang, H. (2015). Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Transactions on Networking (TON)*, 23(2):603–614.

Xiao, J., Xu, Z., Huang, H., and Wang, H. (2012). A covert channel construction in a virtualized environment. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 1040–1042. ACM.

Xu, L. (2010). Securing the enterprise with intel aes-ni. *Intel Corporation*.

Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., and Schlichting, R. (2011). An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 29–40. ACM.

Xu, Y., Cui, W., and Peinado, M. (2015). Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE.

Yarom, Y. and Falkner, K. (2014). Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732.

Yarom, Y., Ge, Q., Liu, F., Lee, R. B., and Heiser, G. (2015). Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015:905.

Yarom, Y., Genkin, D., and Heninger, N. (2017). Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112.

Younis, Y. A., Kifayat, K., Shi, Q., and Askwith, B. (2015). A new prime and probe cache side-channel attack for cloud computing. In *Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT/IUCC/DASC/PICOM), 2015 IEEE International Conference on*, pages 1718–1724. IEEE.

Zhang, S., Zhao, C., Wang, S., and Wang, F. (2017). Pseudo time-slice construction using a variable moving window k nearest neighbor rule for sequential uneven phase division and batch process monitoring. *Industrial & Engineering Chemistry Research*, 56(3):728–740.

Zhang, T., Zhang, Y., and Lee, R. B. (2016a). Cloudradar: A real-time side-channel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 118–140. Springer.

Zhang, Y., Huang, Q., Ma, X., Yang, Z., and Jiang, J. (2016b). Using multi-features and ensemble learning method for imbalanced malware classification. In *Trustcom/BigDataSE/ISPA, 2016 IEEE*, pages 965–973. IEEE.

Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. (2011). Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE Symposium on Security and Privacy*, pages 313–328. IEEE.

Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. (2012). Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316. ACM.

Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. (2014). Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003. ACM.

Zhang, Y. and Reiter, M. K. (2013). Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 827–838. ACM.

Zhang, Z. and Chang, J. M. (2014). A cool scheduler for multi-core systems exploiting program phases. *IEEE Transactions on Computers*, 63(5):1061–1073.

Zhou, Z., Reiter, M. K., and Zhang, Y. (2016). A software approach to defeating side channels in last-level caches. *arXiv preprint arXiv:1603.05615*.